FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

PLATFORM FOR GENERATING AND DEPLOYING WEB APPLICATIONS AND REST APIS

LICENSE THESIS

Graduate: **Florin Adrian IONCE**

Supervisor: **Asis. Ing. Cosmina IVAN**

2017

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

DEAN,
Prof. dr. eng. Liviu MICLEA

HEAD OF DEPARTMENT,
Prof. dr. eng. Rodica POTOLEA

Graduate: **Florin Adrian IONCE**

PLATFORM FOR GENERATING AND DEPLOYING WEB APPLICATIONS AND REST APIS

1. **Project proposal:** The purpose of this project is to define and build a platform able to generate and deploy to the cloud a web application and a REST API with functionalities selected by the users.

2. **Project contents:** Table of Contents, Introduction, Bibliographic Research, Analysis and Theoretical Foundation, Detailed Design and Implementation, Testing and Validation, User's Manual, Conclusions, Bibliography, Glossary, Figures and Tables list.

3. **Place of documentation**: Technical University of Cluj-Napoca, Computer Science Department

4. **Consultants**: Asis. Ing. Cosmina Ivan

5. **Date of issue of the proposal:** February 1st, 2017

6. **Date of delivery:** September 8th, 2017

Graduate: _____

Supervisor: _____

FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

Declaraţie pe proprie răspundere privind autenticitatea lucrării de licenţă

Subsemnatul(a) **Ionce Florin Adrian**, legitimat(ă) cu cartea de identitate seria MM nr. 561079
CNP 1930816240011, autorul lucrării **Platform for generating and deploying web applications and REST APIs** elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la Facultatea de Automatică și Calculatoare, Specializarea Calculatoare din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea Septembrie a anului universitar 2016 - 2017, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data _____

Nume, Prenume _____

Semnătura

Table of Contents

Chapter 1. Introduction 1
1.1. Project context 2

Chapter 2. Project Objectives 3
2.1. Functional requirements 3
2.1.1. Browsing functionalities 4
2.1.2. Selecting functionalities 4
2.1.3. Generating the application 4
2.1.4. Inspecting the application 5
2.1.5. Updating the application 5
2.2. Non-functional requirements 5

Chapter 3. Bibliographic Research 6
3.1. Cloud concept 6
3.1.1. The public cloud model 6
3.1.2. The private cloud model 6
3.1.3. The hybrid cloud model 6
3.1.4. IaaS – Infrastructure as a Service 7
3.1.5. PaaS – Platform as a Service 7
3.1.6. SaaS – Software as a Service 8
3.2. Publishing a web application 8
3.2.1. Development environment 8
3.2.2. Staging environment 9
3.2.3. Production environment 9
3.3. Virtualization with Docker 9
3.3.1. Virtualization 9
3.3.2. Docker Virtualization Tool 9
3.3.3. Docker Containers vs Virtual Machines 9
3.4. Similar systems 11
3.4.1. Heroku 11
3.4.2. Wordpress 12
3.4.3. Prismic 12
3.4.4. Squarespace 13
3.4.5. Conclusions 14

Chapter 4. Analysis and Theoretical Foundation 16
4.1. Platform overview 16
4.2. Server provisioning module 17
4.2.1. Operating principles for the server provisioning module 17
4.2.2. Main responsabilities of the server provisioning module 18
4.2.3. Process of the server provisioning module 18
4.3. Registration / Login page 19
4.4. Functionalities listing page 19
4.5. Application generator module 19
4.5.1. Operating principles of the application generator module 20
4.5.2. Main responsabilities of the application generator module 20
4.5.3. Process of the application generator module 20

4.6. Application editor module *21*
4.7. Use case specification *22*
4.7.1. Browse functionalities 22
4.7.2. Create application 22
4.7.3. Update the application 23
4.7.4. Inspect application 24
4.7.5. Register 25
4.8. Conceptual architecture *25*
4.8.1. MVC Layer 26
4.8.2. Business Logic Layer 27
4.9. User Interface Wireframing *27*
4.10. Technologies used *28*
4.10.1. Ruby on Rails Framework 28
4.10.2. HAML 30
4.10.3. Redis 30
4.10.4. Nginx 30
4.10.5. Docker Cloud 31
4.10.6. WebSockets 32

Chapter 5. Detailed Design and Implementation 33
5.1. Graphical User Interface (GUI) *33*
5.2. Architecture *35*
5.3. Database Design *38*
5.4. Modules Implementation *39*
5.4.1. Server Provisioning module – Docker Cloud Module 39
5.4.2. App Generator Service 40
5.4.3. Update App Service 41
5.5. Application flow *41*
5.6. Generated application *42*
5.6.1. Overview 42
5.6.2. Implementation details 42
5.6.3. Examples 43

Chapter 6. Testing and Validation 47
6.1. Unit testing *47*
6.2. Automated testing *50*

Chapter 7. User's manual 53
7.1. Installation *53*
7.1.1. Prerequirements 53
7.1.2. Install the application 54
7.2. User manual *55*

Chapter 8. Conclusions 59
8.1. Current Results *59*
8.2. Further Development *59*

Bibliography 61

Appendix 1. Glossary 63

Appendix 2. Figures and Tables list 64

Chapter 1. Introduction

Since the apparition of computers more and more people started using them. If at the beginning they were only used for research purposes, this aspect changed in time and now they are used at a large scale for personal purposes too. Together with computer evolution people invented computer networks, which allows them to communicate with each other. One such network is the internet which is in fact the largest and most used network today having over 3 billion users in 2015 with a growth of more than 2 billion in the last 10 years as is shown in paper [5] and having so many users there is a constant request for providing content. More and more users are trying to provide content over the internet which can be accessed using a computer, a smartphone or any *Internet of Things* device but in general creating a web application requires coding knowledge.

A web application is an application running on a web server and can be accessed by the users through a web browser. The communication is established through the HTTP protocol (*HyperText Transfer Protocol*) and is a request – reply based protocol. A web application is composed from two main parts: the back-end and the front-end also known as the server and the client.

The back-end is the application that runs on the web servers and responds to the client's requests by executing a series of operations. The logic of the application is usually implemented on the server but since the evolution of the web browsers some functionalities are implemented directly on the front-end. The back-end application also insures the communication with the database.

The front-end application is the application running on the client side, usually in a web browser. It sends user's requests to the server and displays the response from the server. A front-end application must have an HTML (*HyperText Markup Language*) document in order to define the elements of the application and it usually have CSS (*Cascading style sheets*) documents in order to style the HTML documents. Another important component of a front-end application is represented by the Javascript files. Javascript is a programming language which allows interaction with HTML elements, dynamic styling of the elements but it is mostly used to dynamically change elements from HTML pages and to communicate with the back-end, synchronous or asynchronous - without the need to reload the page through an AJAX request.

In addition to web applications, on the internet there are also SaaS (*Software as a Service*) and PaaS (*Platform as a Service*) through cloud computing. A software as a service offers an application ready to use, without the need of installing or developing anything. A platform as a service provides computing platforms, programming language execution environment, database and web servers for the user to use.

The Internet of Things (*IoT*) is a system of computing devices, objects or digital machines provided with the ability to transfer data over a network without requiring

human to human or human to computer interaction. Most of them are able to connect directly to the internet using RESTful services.

1.1. Project context

Creating web applications or REST APIs providing a series of functionalities is a major problem because both of them require programming knowledge, database knowledge, server configuration skills and the ability to write deployment scripts. The system described in this paper is meant to provide an online platform which offers the possibility to create and deploy fully functional web applications and REST APIs through a web interface, all the configurations and settings are made easily with a sequence of clicks. All the settings, and configurations are created automatically in background, and the user receives a new application with its own database running on a dedicated server.

Chapter 2. Project Objectives

The objectives of the project can be divided into two categories: providing the possibility to create a web application without coding, and providing a REST API which can offer the functionalities of their app, without having to implement them, or to take into account some details like database, server provisioning, server configuration and so on.

Also, those two categories may be combine and provide a web application and a REST API.

2.1. Functional requirements

The implementation objectives include selecting the application functionalities and the method to use them and are made of the following main scenarios:

- **Browsing functionalities –** this scenario allows the users to browse a list with available functionalities, read a description about them.
- **Selecting functionalities –** this scenario allows the users to select the functionalities from a list of functionalities and the way they work – if it used as a web application, an API or both.
- **Generating application –** after the scenarios above are completed the user can generate the application, and at this point the user will be notified by email when the application is available.
- **Sign up –** the users will be able to create and personalize a profile for using this application. They will be using a combination of an email address and a password in order to sign up.
- **Authenticate –** all the steps from the process of generating a new application require a valid account and the user needs to be authenticated.
- **Inspecting the application –** the user can see real time statistics about the application, the configuration which is running, a documentation for the API generated.
- **Updating the application** – this scenario allows the user can update the configuration of the application at any time, or it can even scale the servers which run the application.

The functional requirements are captured in figure 2.1 in a basic use case diagram.

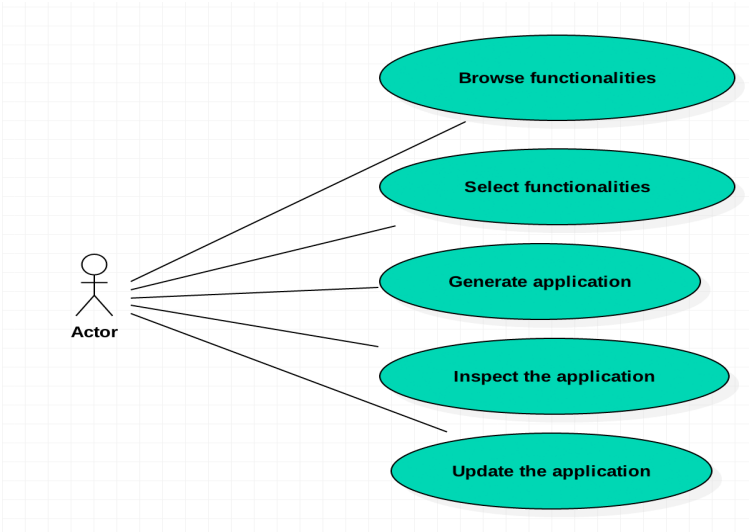


Figure 2.1 Basic use case diagram

Furthermore, we will describe in more details the most important actions of our platform: browsing and selecting functionalities, generating the application, inspecting and updating the application.

2.1.1. Browsing functionalities

This scenario gives the users an overview about the capabilities of the platform, also this is the step where they can decide if they can use it in order to launch the application they need. For this they only need a web browser and a valid account on the platform. They can browse through the list of functionalities, but also, if they want, the can access more information about any of them, and see some examples.

2.1.2. Selecting functionalities

This scenario naturally comes after the users already browsed the list of functionalities, so they start generating a new application, and the first step is to select the functionalities of the application. The users of the system will be able to do this by simply selecting the functionalities using multiple inputs on a form. It is really important at this step that the options are stored in the database, because in case something goes wrong at the next steps the users don't have to start from the beginning.

2.1.3. Generating the application

This is the final step from the flow of generating a new application. In this step the users will basically trigger a task for generating a new application, and they will receive a message saying that they will be notified by email when the setup of the application is completed, since there are more operations executed in background and some of them can take some time.

The email that is being sent contains information about the status of the application, if it was generated successfully or if there occurred some errors. Also it contains a link to the new application, from where it can be accessed, an admin account, if needed. If the application is a REST API, there will be an access token. After this scenario, the users will be able to access and use the new application.

2.1.4. Inspecting the application

This scenario allows the users to inspect real data statistics about the generated app. The statistics include the services running on the server, the status of the application and the link to access it directly.

If the generated application contains REST API, a documentation is available with examples for each request possible and a list with all the parameters accepted by each end point.

2.1.5. Updating the application

This scenario is similar to the *Selecting functionalities* scenario, the main difference being that in this case the application is updated. The user can change all the functionalities, scale the application (add a stronger CPU, more RAM memory).

An important aspect of this step is that none of the data from the database is lost at the update operations.

2.2. Non-functional requirements

- Usability – User friendly, easy to use
- Accessibility – The generated application should be public, available for anyone to use
- Stability – Uptime, restart in case of failure
- Documentation – How to use the generated API
- Performance – Generate application should have a low response time, the ability to serve multiple concurrent users

Chapter 3. Bibliographic Research

This chapter will describe the main concepts – cloud and publishing a web application. Also it presents an analysis of a set of similar systems considering that there are several solutions to publish web applications even without coding.

3.1. Cloud concept

The concept of cloud represents a service which offers hardware and software resources to the users. Since most of the internet users have a permanent and fast connection we now have the possibility to share resources through the internet and access them only when they are necessary. The paper *Types of Cloud Computing* [1] states that the cloud computing deployment models can be of 3 types: public, private and hybrid.

3.1.1. The public cloud model

The public cloud model allows a service provider to offer resources such as applications and storage, available to the general public over the internet. Public cloud services can be free or offered on a pay-per-usage model.

3.1.2. The private cloud model

Private cloud delivers similar advantages to public cloud, including scalability and self-service, but they are using a proprietary architecture. Private cloud is dedicated to a single organization and is best for business with dynamic computing needs that require direct control over their environments.

3.1.3. The hybrid cloud model

Hybrid cloud computing is a mix between private and public cloud, by allowing workloads to move between public and private cloud as computing needs and costs can change. Hybrid cloud gives businesses greater flexibility and deployment options.

In addition, there are 3 types of Cloud Computing Models: IaaS *(Infrastructure as a Service)*, PaaS *(Platform as a Service)* and SaaS *(Software as a Service)* illustrated in figure 3.1. It can be noticed that each cloud computing model can be considered a layer on top of the previous one.

[1] Types of cloud computing, https://aws.amazon.com/types-of-cloud-computing/

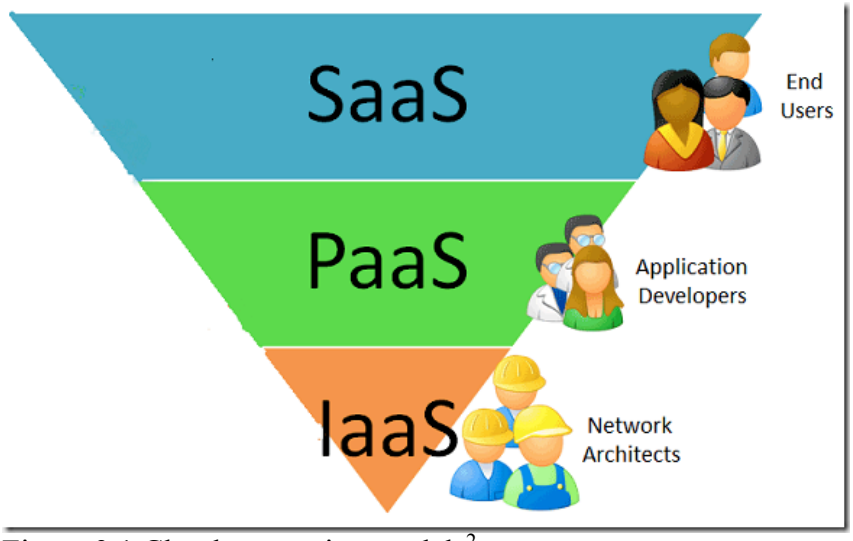


Figure 3.1 Cloud computing models[2]

3.1.4. IaaS – Infrastructure as a Service

Infrastructure as a Service contains the basic blocks for cloud computing and it usually provides access to networking features, computers (virtual or using dedicated servers) and data storage space. It can be observed in figure 3.2 that infrastructure as a service is the lowest layer from the cloud computing layers.

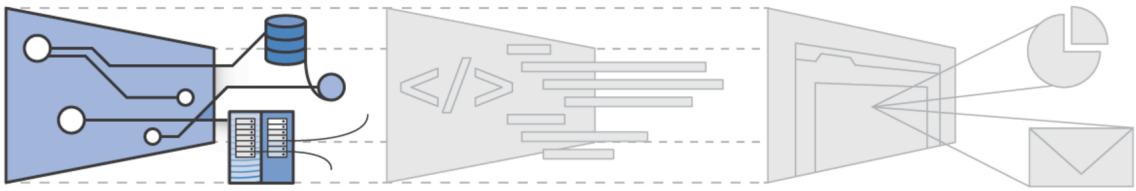


Figure 3.2 Infrastructure as a service layer[2]

3.1.5. PaaS – Platform as a Service

Platform as a Service allows users to focus on deploying and managing the applications without the need of managing and organize the underlying infrastructure. This allows users to be more efficient since it's solving resources planning and software maintenance issues. This is the middle layer of the cloud computing models and it's illustrated in figure 3.3

[2] Cloud computing outlook - https://qarea.com/articles/cloud-computing-outlook-iaas-pa as-and-saas

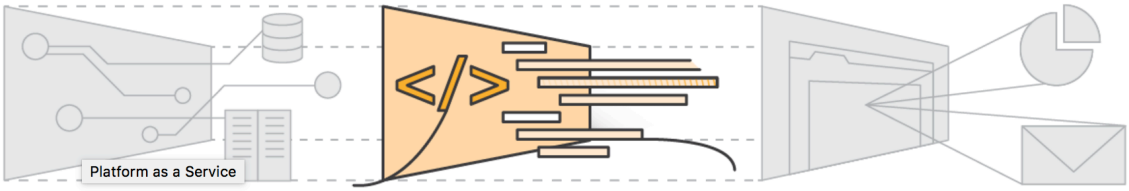


Figure 3.3 Platform as a service layer[2]

3.1.6. SaaS – Software as a Service

Software as a Service provides a complete, final product which is managed by the service provider. They are also called end-user applications and they offer to the user only the software, without revealing the way it works and how it is managed. This layer is illustrated in figure 3.4.

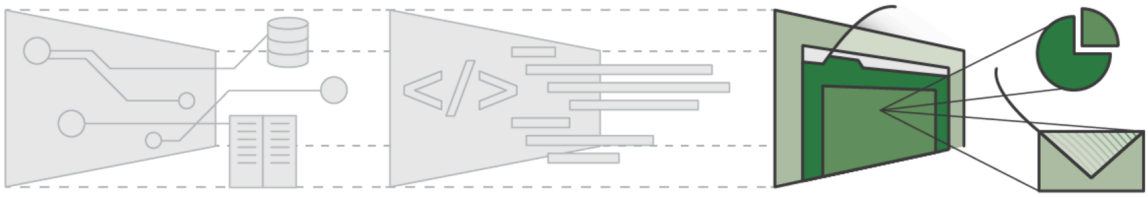


Figure 3.4 Software as a service layer[2]

3.2. Publishing a web application

Publishing a web application or deploying it contains all the processes involved in getting new software and hardware resources running in its environment and becoming available for the targeted users. The deployment processes include installation, configuration, running and testing the application on the new environment. Those scripts can be automated, or can be used several tools, such as Ruby on Rails's Capistrano or Mina deploy which allows users to create scripts and configuration which then automate the deployment process.

There are usually 3 environments, as mentioned in article [7] when developing a web application: development, staging and production.

3.2.1. Development environment

Web applications don't need a remote environment for development, each developer can have the application installed locally. In this case the deployment process is simpler, because development web servers are usually less complex and the app still runs locally, it does not need to be deployed remotely. An example of such a deployment process would be Ruby on Rails web server called Webrick.

3.2.2. Staging environment

This is an intermediary step between developing an application or a feature and releasing it to the users. This environment is remote, usually on a dedicated server and it is very important to have similar configurations like the production environment, thus minimizing the risk of bugs that can not be reproduced on staging environment. This is the environment where all the tests should be made, and no release can be made until all the tests have passed on staging environment.

3.2.3. Production environment

This is the environment used by the users directly, and every feature deployed here must be tested before on staging environment. All features should be developed starting from the code which runs in production thus eliminating the conflicts and unstably when releasing a new feature.

3.3. Virtualization with Docker

3.3.1. Virtualization

The paper *Orchestration of smart objects with MQTT for the Internet of Things* [3] states that the server virtualization technique (also known as *Hypervisor)* is the base technique used in base computing and there are multiple virtualization tools used for cloud computing.

3.3.2. Docker Virtualization Tool

Docker is an application level virtualization software [2], using *Libcontainer* [4] (a linux container system) used to run and manage applications in isolated containers. It automates the processes of configuring development, staging and production environments. All the the complexity is managed by docker, once an app is dockerized.

3.3.3. Docker Containers vs Virtual Machines

Containers are more performant and lightweight than virtual machines [5] because they don't bundle an entire operating system, they are using resources from the host. A comparison between virtual machines and docker containers is illustrated in figure

[3] G. Nalin, "Orchestration of smart objects with MQTT for the Internet of Things", University of Padua, Department of Information Engineering, Master Degree in Computer Engineering

[4] Docker and broad industry coalition unite to create open container project - https://blog.docker.com/2015/06/open-container-project-foundation/

[5] Overview of what docker is - https://www.docker.com/what-docker

3.5.

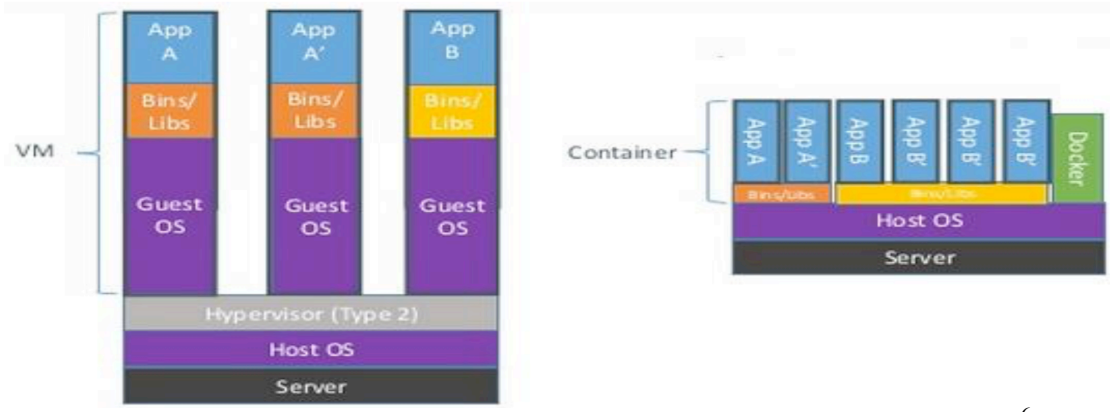


Figure 3.5 Comparison between virtual machines and docker containers[6]

Besides the fact that the storage needed by docker containers is smaller than the storage needed by the virtual machines, there is also a significant difference on boot time and computation time. Figure 3.6 shows us the boot time difference between docker machines and kernel-based virtual machines (*KVM*) using the same hardware and software resources – 20 images and the performance was measured with NMON tool[6].

Since the containers are using the operating system from the host machine, there is no need to boot an entire operating system when lunching a new image, it's just a matter of booting resources which are not included in the host system's packages.

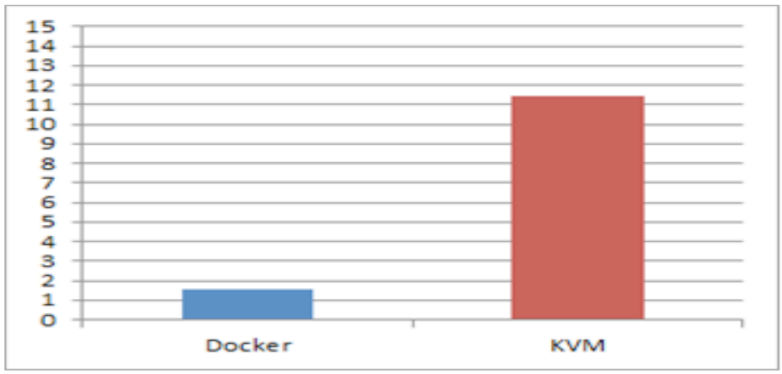


Figure 3.6 Docker vs KVM boot time measured in seconds[6]

Also, the CPU computing power between docker containers and virtual machines was measured in paper [2], there were two tests – calculating 100000! and 200000! In python. Both of the results show that containers are slightly faster. The results can be observed in figure 3.7.

[6] Performance monitor tool for linux - http://nmon.sourceforge.net/pmwiki.php

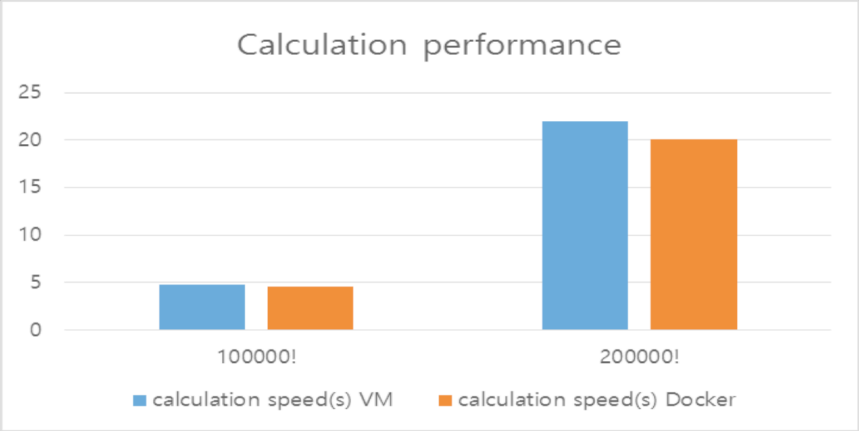


Figure 3.7 CPU computing power[6]

3.4. Similar systems

The main purpose of this paper is to develop a tool which allows users to create a new application without having to implement it. In order to achieve this purpose, we will have to implement a cloud platform which will list allow them to configure and generate applications. The platform will also be a web application, and the generated software will be stand-alone applications, on different machines, without any dependencies from the platform.

There are several platforms offering similar functionalities. In this section we will take a look at them, analyze what they offer and what are the differences between them and our system.

3.4.1. Heroku

Heroku [8] is a cloud platform that allows users to build, deliver, scale and even monitor applications. The don't actually deliver applications without coding, but they are offering a platform for deploying and configure applications without focusing on the infrastructure layer.

Their goal is to allow users to focus on the application only, and spend time developing application and launch it in production as soon as possible.

Heroku allows users to run and manage applications written in Ruby, Node.js, Java, Python, Clojure, Scala, Go and PHP. They also offer support for some frameworks commonly used with those programming languages like Ruby in Rails or Django. They don't need all the source code of an application in order to install the dependencies, they know how to manage them using files like *Gemfile* in Ruby on Rails, *package.json* for Node.js or a *pom.xml* in Java. They also allow users to run other application, by using Procfiles. Procfiles lets them explicitly mention what operations they want to run.

Deploying application on Heroku is made with git, which is a distributed version control system used to manage versions of source code. The Heroku platform uses git as the primary means for deploying applications. When the users create a new application on Heroku it is associated automatically with a new git remote, usually named *heroku* with the local git repository for the application. Basically after the setup is made the deploy is created with basic *git push* command to the *heroku* remote. Of course there are other ways to deploy an application to Heroku, like Github Integration, such that each pull request is associated with a new application. There is also Dropbox Sync, which deploys the application from a Dropbox directory. Finally, there is also a Heroku API for building and releasing applications.

After an application is deployed on heroku, it is initiated the build mechanism. It is specific for each programming language, but the main steps are the same, following the same pattern. Usually the new dependencies are installed, some files and assets are generated and then the application is launched.

3.4.2. Wordpress

Wordpress[7] is an online platform mostly used for creating websites or blogs. It is written in PHP, uses a MySQL database and it has a complex content management system.

Wordpress can be downloaded and installed on a self-hosted server, but it can also be hosted on a using one of their own hosting services. It can be customized using existing templates and plugins written mostly in HTML, CSS, PHP and javascript, but of course, anyone can create new templates and plugins.

Although wordpress inspired my project to use isolated functionalities that can be added in modules and can work individually, they don't support by default the ability to create REST APIs. There is the possibility to create REST APIs [8] using wordpress but it does not come out of the box and it still requires knowledge about how their platform works.

3.4.3. Prismic

Prismic [9] is a self-hosted content management system which publishes content inserted by the users using a rich text editor. The content is published in JSON format available through HTTP requests. In figure 3.8 it is presented the way Prismic data is configured and how the output defined.

[7] About wordpress - https://wordpress.org/about/

[8] Wordpress plugin for generating REST APIs - https://wordpress.org/plugins/rest-api/

[9] Content management system - https://prismic.io/

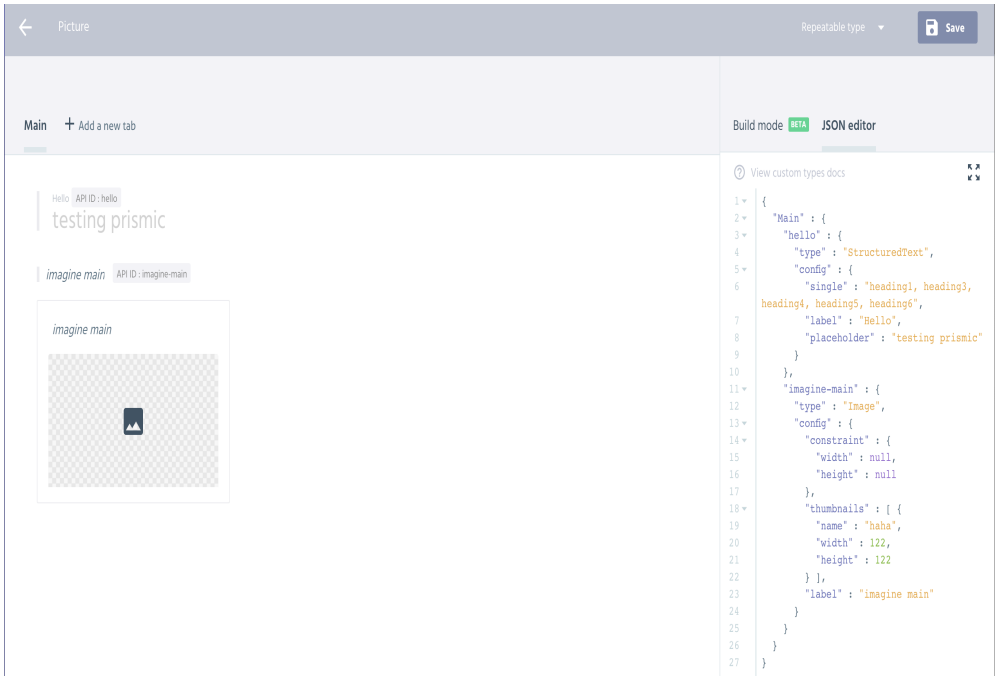


Figure 3.8 Prismic content configuration and output preview

Unlike my platform, prismic's API can be used only to read data which was inserted from a web page, so the API available by them can not be used to create, update or delete content and can't be used to create full applications.

3.4.4. Squarespace

Squarespace is a software as a service specialized on website building and content management for non-technical users. Their system allows users to build websites, online stores and even mobile applications. Their configuration method is called *what you see is what you get* meaning that the users configure exactly the way the website works, unlike other systems like wordpress or Prismic where the user adds data in in a standardized format and he needs to configure or implement separately the way the content is displayed.

Figure 3.9 Building a squarespace page [10]

In figure 3.9 is an example of how a website is built using squarespace's editor and the fact that hovering over any element from the webpage allows the user to edit its content.

3.4.5. Conclusions

The table 1 shows a comparison between the similar systems and our platform, based on the following criteria:

- Infrastrucutre – If it's self hosted or it provides a hardware infrastructure
- Use as API – If the product provided can be used as an API from web or mobile applications or even *IoT* devices
- Website – if the system / platform can be used to obtain a website
- Self generated – If the user can use a product without coding or there is a need to build an application
- Configurable hardware resources – If the user can choose a hardware configuration.

It can be noticed that the least common feature on the systems is the ability to use tha product as an API, so it can be used from multiple devices. Only *Prismic* and our proposal having these features.

Also, the ability to choose the hardware configuration based on the needs it's quite rare, only *Heroku* and our platform are providing this feature, but since *Heroku* it's only an *infrastructure as a service* it is not able to generate applications without coding.

[10] Configuring a squarespace page - https://florin-ionce.squarespace.com/config/pages

Table 1 Similar systems conclusions

	Heroku	Wordpress	Prismic	Squarespace	Our System
Infrastructure	Yes	No	Yes	Yes	Yes
Use as API	No	No	Yes	No	Yes
Self generated	No	Yes	Yes	Yes	Yes
Website	No	Yes	No	Yes	Yes
Configurable hardware resources	Yes	No	No	No	Yes

To sum up, there are systems which provide some of the functionalities of the proposed platform, but, based on our knowledge, there is no universal solution for both generating and deploying on a dedicated server, with control of the server's capabilities at a hardware level which generates a web application and a REST API without any other configuration or extra work.

Chapter 4. Analysis and Theoretical Foundation

4.1. Platform overview

Based on the specifications, requirements and objectives defined in Chapter 2, a basic flow of the platform can be defined as follows in figure 4.1.

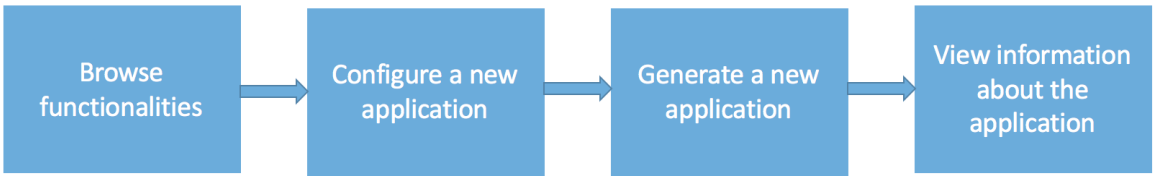


Figure 4.1 Basic platflorm flow

Considering the basic platform flow, we can see that at each step, there is a strong interaction between the user and the data extracted from the database. By using the MVC (Model – View - Controller) architecture for the main platform most of the non – functional requirements can be achieved.

The only addition to the architecture would be a background processor for the non-performant operations. This layer would contain all the business logic for generating a new application. The business logic layer should still have interaction with the database and the models, but not with the user directly. Those constraints indicate the use of an architecture illustrated at a high level in figure 4.2, with user having interactions only with the views layer, which send all the response to the controller. The controller should decide which of the operations must be processed synchronous or asynchronous and it should also be the layer which renders data back to the views, in order to be accessible for the users.

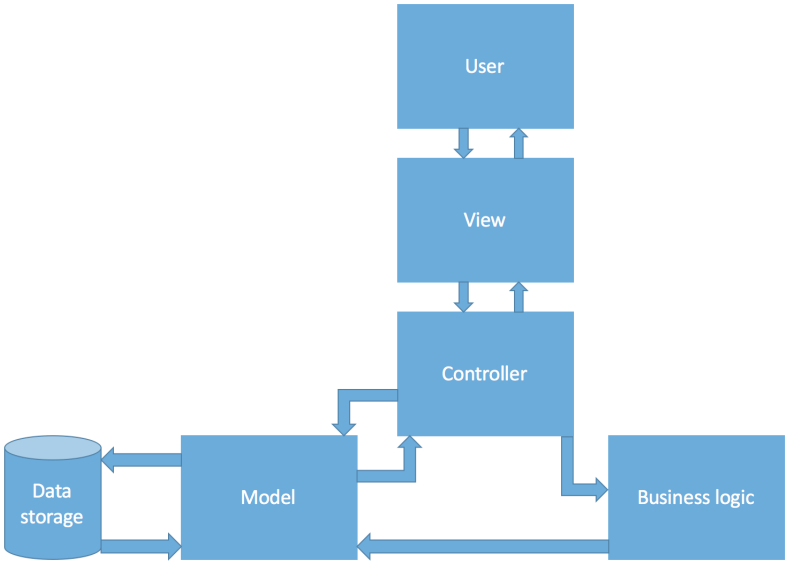


Figure 4.2 High level project architecture

Based on the previous analysis the platform should be broken down into multiple modules

- Server provisioning module
- Registration / Login page
- Functionalities listing page
- Application generator module
- Application editor module

In the following sections we will begin the concepts and the operating principles and concluding with the conceptual architecture of the system.

4.2. Server provisioning module

4.2.1. Operating principles for the server provisioning module

This module should handle all the operations and resources necessary for installing all the dependencies needed by a newly created application in order to run. It should not handle any of the internal business logic for the generated application or for the platform which generates the applications.

In ideal case, this module should be completely isolated from the rest of the platform following the single responsibility principle – installing all the dependencies based on input data, without the need to perform any other operations. However, in order to ensure data persistency, it may need to store data in the database used by the platform because the user should able to see the progress of the new application in real time.

The best approach would be not to access the database directly from the Server provisioning module, since the platform's components should not be coupled in this manner. As illustrated in figure 4.2 this module would only communicate to the models or to a middleware between them. This solution would be more flexible and scalable.

Considering the conclusions from Chapter 3 section 3, for this service a virtualization tool should be used – more precisely Docker, because it offers the capability to install all the dependencies on any server running linux. Once the base application is docker compatible, there is no need for extra work to launch a new instance for a user.

Also, using docker offers flexibility when choosing a server provider, because it can be run on most of the providers, starting from Amazon's AWS[11], to Digital Ocean[12] or Google Cloud platform[13].

[11] Amazon docker support - https://aws.amazon.com/docker/

[12] Digital Ocean docker support - https://www.digitalocean.com/products/one-click-apps/docker/

[13] Google cloud platform docker support - https://cloud.google.com/compute/docs/instance-groups/deploying-docker-containers

Docker also offers support for scalability and load balancers[14], which makes it a good choice for a self-managed app, considering that the users are receiving software as a service, they should not have to interact with any of the processes of scalability and load balancing of the application.

4.2.2. Main responsabilities of the server provisioning module

- Launch new server with the configuration requested by the user
- Install dependencies needed by the application in order to run
- Install the application requested by the user
- Notify the main platform with the progress at any point

4.2.3. Process of the server provisioning module

As described in figure 4.3, the server provider module should be initialized with user's preferences regarding the application configuration (CPU power, geographic location) and the functionalities selected, because the new servers should be configured with respect to user's selections.

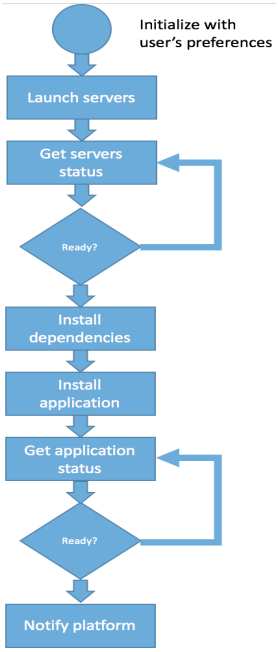


Figure 4.3 Server provider module state diagram

[14] Docker load balancing solution - https://docs.docker.com/docker-cloud/getting-started/deploy-app/9_load-balance_the_service/

The next steps would boot the requested servers and install all the dependencies. After all the dependencies are installed, the user's application can be launched and the main platform can be notified about it.

4.3. Registration / Login page

Those pages represent the initial steps for the user. It should only contain some generic information on how the application works and it should allow the user the ability to join the application and get access to the application in order to be able to proceed with the steps for generating a new web application or a REST API.

Figure 4.4. illustrates a mockup of the content needed for this step. In addition to those information, those forms for register and login should communicate with the data base and have the ability to perform queries, validate data and give access to the users.

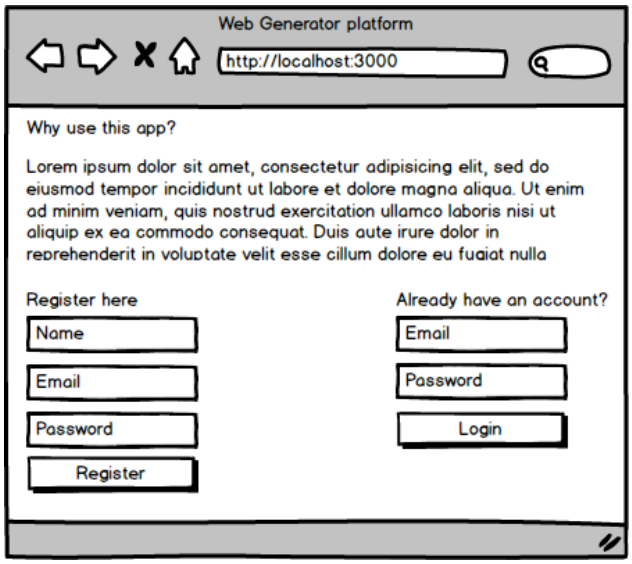


Figure 4.4 Register / Login mock-up

4.4. Functionalities listing page

This page should contain all the information needed by the user in order to understand all the capabilities of the platform and how it should be used.

It should not have a strong impact on the architecture, technologies, performance or implementation of the project but it should contain clear information, with examples, even illustrations.

4.5. Application generator module

This, together with the service provider module is the most important parts of the platform.

4.5.1. Operating principles of the application generator module

It should should have a strong interaction with the users, through a web page, where the user has the possibility to configure the application desired. This means that it should contain a web form, inputs containing all the options available with labels to describe the role of each input.

When the user configures the application and choses to generate it, this module should store all the preferences in a database, collect all the relevant inputs for the server provisioning module and launch it.

It can be observed that this module should interact with most of the parts from the application, reading user's input, storing all the preferences to a database, performing validations on the data and launching a service which would generate a new application for the user.

In order to fulfill the non-functional requirements mentioned in Chapter 2, all the data must be validated on the back-end, on the server side and it must be stored in the database. Considering that performance and usability are important requirements, the validations must also be performed on the front-end application.

4.5.2. Main responsabilities of the application generator module

- Display configuration options for the user
- Perform front-end validations on user's input
- Submit user's configuration to the back-end
- Perform backend validations
- Display precise error messages if needed
- Store user's configuration
- Launch server provisioning module

4.5.3. Process of the application generator module

As presented in figure 4.5, the application generator module would read and display the available configuration to the user, validate any input data, and submit the form to the back-end when the data is valid.

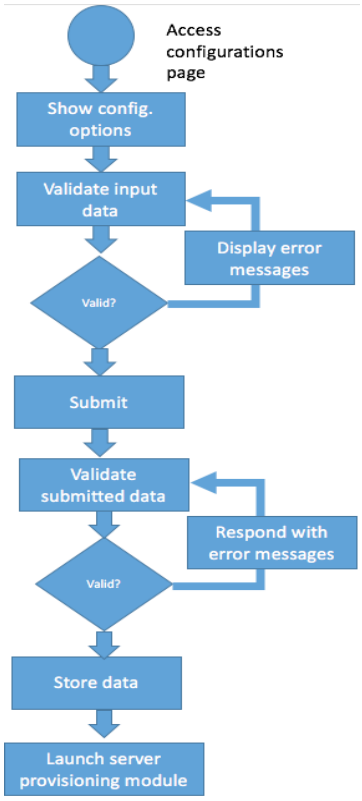


Figure 4.5 Application generator module state diagram

When the data is submitted to the back-end, there is another round of validations and only if the data is valid the selections are stored into the database and the server provisioning module is launched, otherwise the server should respond with error messages.

4.6. Application editor module

This module should allow a user to update both the application functionalities and the server capabilities like RAM memory, location, CPU power or storage.

This means that most of the components should already be implemented so in order to avoid duplicating code all the modules presented before should be reusable such that the edit action is available.

A generic use case would be that the user has a button to edit the generated application, and a form similar to the one from creating new applications should be rendered. The process of updating should still be performed asynchronous by the server provisioning module because, in theory it should already have a middleware layer to communicate with the chosen server provider.

4.7. Use case specification

The basic application flows were listed in chapter 2 and in figure 2.1, in this section we will provide detailed use cases for each of them.

4.7.1. Browse functionalities

Primary user
Normal User

Description
The users can read about the capabilities of the platform and the types of application which can be generated.

Stakeholders and interests
Non technical users – interested in publishing a web application or a blog
Technical users – interested in building applications without the need of developing the back-end application and managing the infrastructure.

Basic Flow
1. The user clicks the "Browse functionalities" button from the menu.
2. The user scrolls between functionalities and expand them to find more information.

Alternative Flows
-

Special requirements
-

Preconditions
1. The user has a valid account.
2. The user is logged in.

Postconditions
-

4.7.2. Create application

Primary user
Normal User

Description
The users select functionalities and application settings and create a new application.

Stakeholders and interests

Non technical users – interested in publishing a web application or a blog
Technical users – interested in building applications without the need of developing the back-end application and managing the infrastructure.

Basic Flow

1. The user clicks the "Create new application" button from the menu.
2. The user writes the application's name.
3. The user selects the options from the dropdown selectors.
4. The user clicks the "Create" button.

Alternative Flows

1. Wrong data submitted – validation failed
   The user is redirected to the form page and errors are displayed on the fields submitted with wrong data.
2. Application generation failed
   An internal server occurred – the user will be notified via email that there has been an error while generating the app.

Special requirements

-

Preconditions

1. The user has a valid account.
2. The user is logged in.
3. The user does not have another application generated.

Postconditions

1. The user will be able to access the new generated application
2. The user will be notified via email with the credentials and the url of the application

4.7.3. Update the application

Primary user
Normal User

Description
The user updates the functionalities or settings of a previously generated application.

Stakeholders and interests
Non technical users – interested in publishing a web application or a blog
Technical users – interested in building applications without the need of developing the back-end application and managing the infrastructure.

Basic Flow

1. The user clicks the "Edit your application" button from the menu.

2. The user selects the options from the dropdown selectors.
3. The user clicks the “Update” button.

Alternative Flows

1. Wrong data submitted – validation failed
   The user is redirected to the form page and errors are displayed on the fields submitted with wrong data.
2. Application generation failed
   An internal server occurred – the user will be notified via email that there has been an error while generating the app.

Special requirements

-

Preconditions

1. The user has a valid account.
2. The user is logged in.
3. The user has an application generated.

Postconditions

The user’s changes will be reflected in the application.

4.7.4. Inspect application

Primary user

Normal User

Description

The user inspects statistics and details about the application.

Stakeholders and interests

Non technical users – interested in publishing a web application or a blog

Technical users – interested in building applications without the need of developing the back-end application and managing the infrastructure.

Basic Flow

1. The user clicks the “My application” button from the menu.
2. The user opens application address.

Alternative Flows

1. The user reads about the services running.
2. The user opens the API documentation for the generated application.

Special requirements

-

Preconditions

1. The user has a valid account.
2. The user is logged in.

3. The user has an application generated.

Postconditions
-

4.7.5. Register

Primary user
Normal User

Description
The user needs to have a valid account in order to use the platform.

Stakeholders and interests
Non technical users – interested in publishing a web application or a blog
Technical users – interested in building applications without the need of developing the back-end application and managing the infrastructure.

Basic Flow
1. The user clicks on the "Sign up" button from the menu.
2. The user fills in the credentials.
3. The user clicks the "Sign up" button.

Alternative Flows
1. Email already taken – validation failed
   The user is redirected to the registration page and error messages are displayed.
2. Email address is invalid – validation failed
   The user is redirected to the registration page and error messages are displayed.

Special requirements
-

Preconditions
1. The user is connected to the internet.
2. The user has a web browser installed.

Postconditions
1. The user has a valid account.
2. The user can access the other functionalities.

4.8. Conceptual architecture

In the previous sections, all the modules were described individually. In this section they will be combined into the complete system architecture.

As mentioned before, the system architecture will use the MVC pattern and will use other modules as micro services. This pattern is used to separate the views and the controller and to implement the main login in models and services.

In addition to the MVC components, there should be another layer for the business logic and different services which should be extracted from the controllers and models as illustrated in figure 4.6.

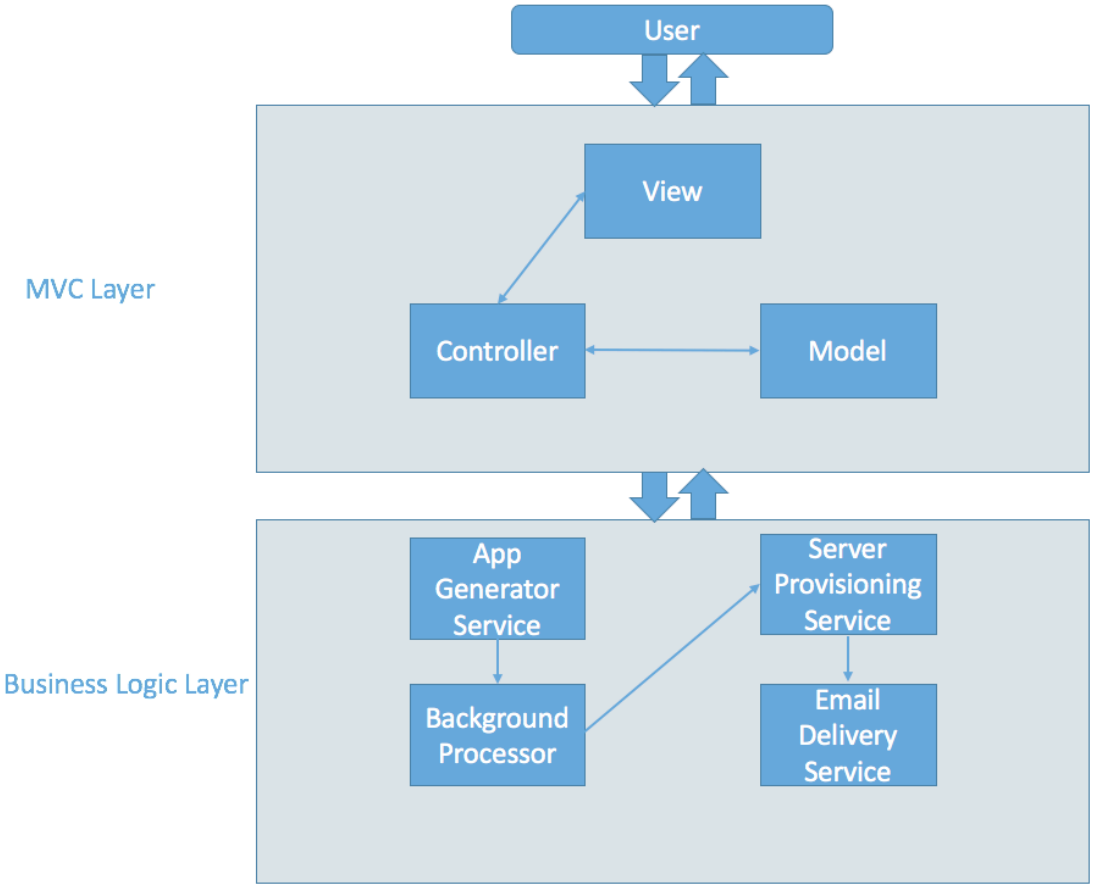


Figure 4.6 Conceptual architecture

4.8.1. MVC Layer

The MVC layer should contain three major modules: The views, the controllers and the models which should have access to the database.

Ideally, the resources accessed by the users directly, like App entity or the User entity should have a model, a controller and views for each actions. Each of the actions available should be defined in the controller for that entity.

For the other entities, like an entity which would store information about the Server's configuration and status there is no need for views and controller since all this

entity needs to to is to store and read data from the database so only the model is required.

The views must not have access to the database, they should only communicate with the controllers by sending HTTP requests and by rendering data obtained from the controller. It will be a request – reply communication.

The controller should be responsible for accepting requests from the views and performing the read or write operations from the models which should be the only ones accessing the database directly. A controller can access models from different resources if necessary.

4.8.2. Business Logic Layer

The business logic layer should contain all the modules, services, algorithms and third party interfaces making the entire system more abstract.

The business logic layer should contain the following modules/services:

- App Generator service
- Background Processor
- Server Provisioning Service
- Email Delivery Service

This layer should be able to access the database, but not directly, but with APIs defined on the models or even REST APIs.

4.9. User Interface Wireframing

Before starting the front-end implementation, some design mock-ups were made in order to facilitate the development process.

The figure 4.7 presents the functionalities page structure, but also the structure of the entire application. Since the menu and other buttons are reviled.

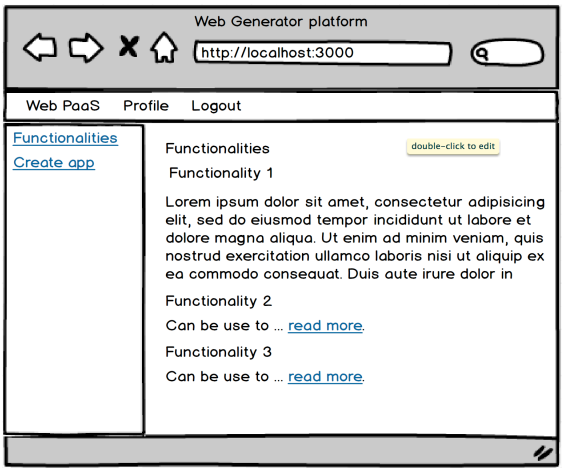


Figure 4.7 Functionalities page mock-up

The figure 4.8 describes the page where a new application is created.

It should contain a simple form with dropdowns for each functionality, where the user should be able to specify it the functionality should be available for the Web application or for the API or both. It can be noticed that some of the functionalities should have extra options, like comments.

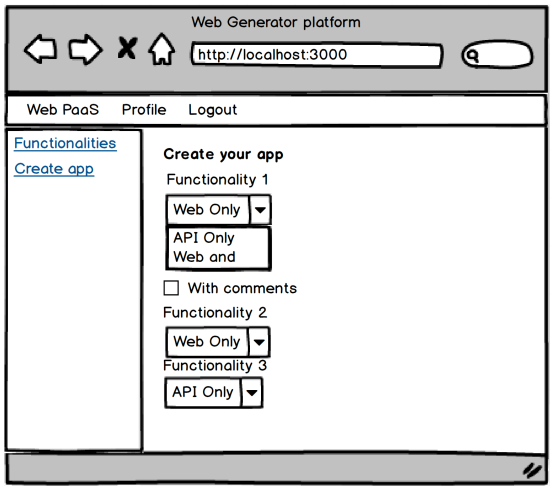


Figure 4.8 Create your application page

The mockup for Update the application page is not necessary since it should be very similar to the Create your application page.

4.10. Technologies used

4.10.1. Ruby on Rails Framework

Ruby on Rails[15] is a server-side web application framework written in Ruby[16] and based on *WEBrick*[17] and *Rack*[18]. Rails is a model-view-controller framework and provides default structures for web services, webpages and databases – figure 4.9. It has build-in mechanisms for JSON or XML data transfer and support for HTML, CSS and JavaScript. Besides the MVC pattern, Ruby on Rails uses the convention over configuration and don't repeat yourself paradigms and it's also one of the first frameworks to use the active record architectural pattern.

[15] Ruby on Rails framework - http://rubyonrails.org/
[16] Ruby language - https://www.ruby-lang.org/en/
[17] WEBrick - http://ruby-doc.org/stdlib-2.0.0/libdoc/webrick/rdoc/WEBrick.html
[18] Rack - https://github.com/rack/rack

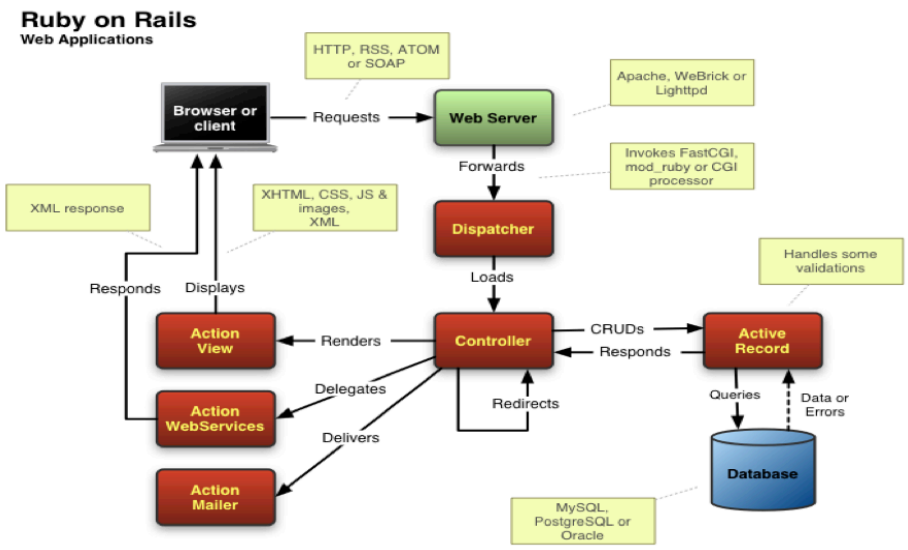


Figure 4.9 Overall framework architecture[19]

In fact, Active Record is one of the core components from the Rails framework and the library which implements the most used ORM library in rails shares the same name. ActiveRecord is also the default connection to the database and it creates persistable domain models from business objects and database tables, where logic and data are presented as a unified package.

Ruby on Rails comes with the following features out of the box:

- Development Server
- Database Connection
- Database ORM
- RESTful request dispatching
- Script generators to automate tasks
- Integrated unit testing support
- ERB templating engine
- SMTP email management
- Defined file structure
- Multiple environments support

All the configuration of the project is stored inside the *config/* folder and it supports multiple environments, they just need to be defined in the *config/enivrionments/* folder.

The main configuration files / options are:

- database.yml – stores the database configuration
- secrets.yml – third party tokens, app secret UUID and other constants

[19] Ruby On Rails Architecture - http://adrianmejia.com/blog/2011/08/11/ruby-on-rails-architectural-design/

- assets.rb - loads the CSS, JS, images and libraries used by the application
- environment.rb – starts all the initializers which boot up the application

4.10.2. HAML

HAML[20] is an *HTML* abstraction mark-up language which is interpreted into *HTML* at run-time but a *HAML* preprocessor. Its main purpose is to keep *HTML* clean and simple and to use *HAML* functions instead of writing inline HTML properties.
It is mostly used by Ruby on Rails applications, but it's also used in *PHP*, *ASP* or static websites build with tools like *middleman*[21].

The main improvements *HAML* brings over *HTML* is that it's less repetitive, more dry because the elements don't have to be name twice, unlike *HTML* where each tag needs to be opened and closed, and they avoid this by relying on indentation, basically everything that is indented inside an element it's contained by it, thus closing tags is not needed anymore. This feature also ensures that the code is well-indented, more organized, unlike *HTML* where the proper indentation is optional. Also it is a lot easier to read for humans because it is less bloated with tags and properties.

4.10.3. Redis

Redis[22] is an open source data structure used as a database. It works with in-memory key-value store and supports multiple data structures such as hashes, lists, sets, sorted sets and it comes with built in master-slave replication.

Persistence is ensured by holding the data in the virtual memory, but from time to time it is also transferred to the disks in *.rdb* format. It can be used as a database, cache system and message broker. In our project, *redis* would be used as a message broker using the *Publish – Subscribe Paradigm*[23] where some of the client's operations would pushed to *redis* which will push to other services called *workers* who are subscribe to certain events and execute different jobs.

4.10.4. Nginx

Nginx[24] is a free, open-source web server, reverse proxy, load balancer and *HTTP* cache, which is known for it's great stability, performance, low resource consumption and the large set of features. It can be deployed to serve dynamic *HTTP* content when

[20] HAML language - http://haml.info/about.html
[21] Middleman tool - https://middlemanapp.com/
[22] Redis db - https://redis.io/
[23] Publish Subscribe - https://en.wikipedia.org/wiki/Publish–subscribe_pattern
[24] Nginx reverse proxy - https://www.nginx.com/resources/wiki/

used with application servers like *uicorn*[25] or *phusion passenger*[26]. Without an application server, it can still be used to serve static pages and assets.

It is using a lot less memory then *Apache web server*[27], and can handle about four times more requests, according to paper [19]. However, this performance increase comes with less flexibility and less capabilities, out of the box but it is using a modular architecture and the latest versions support dynamic loading of modules which can solve most of the issues created by this compromise.

4.10.5. Docker Cloud

Docker cloud[28] provides hosted service where *dockerized* application images can be run in order to be distributed to a cloud service provider or to our own linux servers. It facilitates operations for deploying web applications, managing host infrastructure. It can store pre-built images or it can be linked to a source code and build the docker images from it, if the code it's *dockerized* properly.

For the infrastructure management, in order to perform operations like running tests on am image, or building a new one from sources, docker cloud either provides the needed infrastructure or it can be setup on our own infrastructure. They also have the ability to bring our own nodes integrated in their infrastructure, without the need to install *docker cloud* on one of our self-hosted servers.

In terms of accessibility, it can be used either from a web interface, but this does not allow automation, or through an API, which can be used to perform all the operations available for creating, deploying, scaling or maintaining the applications and it can also be automated. In addition to those two methods, docker cloud can also me bused through a *command line interface* which is really helpful when building and maintaining applications and it can also be used for automating certain processes.

In terms of automation, we've covered the two options and they both have full capabilities on *docker cloud* infrastructure. However, the *command line interface* method was rather build to be used by an operator as for the API, it can be easily used from another application since it's a *RESTful API,* highly documented and with great performance.

[25] Unicorn - https://github.com/blog/517-unicorn
[26] Phusion passenger- https://www.phusionpassenger.com/
[27] Apache web server - https://httpd.apache.org/
[28] Docker cloud https://docs.docker.com/docker-cloud/

4.10.6. WebSockets

WebSockets[29] are a communication protocol which makes it possible to open an interactive communication session between a user a server. It allows full duplex communication, which means that on the same channel a user can both receive or send messages. It was designed to work between a web browser and a web server over TCP connection but it can be used by any client-server application. The protocol has been standardized into an API which all the major browsers support called *WebSoclets API*[30].

The *WebSockets API* is easy to use, having only four events: *open, message, error* and *close.* Once the connection is open, the messages are sent real-time unless the connection is closed or an error occurs, in which case, the connection can be opened again. There are plenty of libraries using *WebSockets API* for real-time communication, including *ActionCable*[31] which is ideal for integrating websockets into a *Ruby on Rails* application.

[29] WebSokets Protocol - https://en.wikipedia.org/wiki/WebSocket
[30] WebSockets API - https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API
[31] Action Cable - https://github.com/rails/rails/tree/master/actioncable

Chapter 5. Detailed Design and Implementation

In this chapter we present all the modules implementation and design solutions. The concepts were presented in Chapter 4, and here we present the details of the implementation and the technologies used.

5.1. Graphical User Interface (GUI)

The Graphical User interaface is the main user interaction module of the platform, since all the user actions are handled by it.

The implementation was made using HAML[32] which is a simple markup language compiled into HTML, CSS and Javascript. The base style was given by bootstrap libraries[33].

The user interface is divided in three sections:

- Top navigation bar – contains the following: Home button (brand logo), notifications and settings page link.
- Side navigation bar – contains urls to the functionalities page, create new application page – if no application was created, edit your application page and view your application page - if an application was created.
- Container – organized into panels and sections. The content of the container changes based on the page accessed by the user.

One of the most important pages is the 'Create your application' and it can be seen in figure 5.1 that it is very simple in order to provide usability to the users.

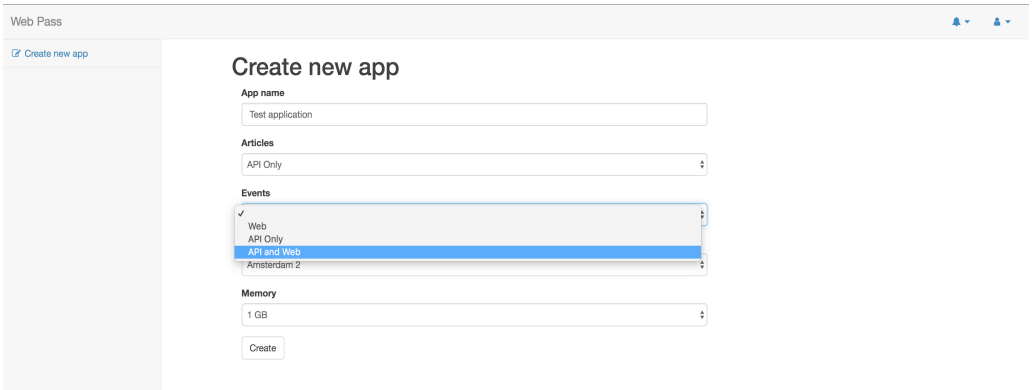


Figure 5.1 Create new application page

Another important page is the 'API Docs' page – which is only accessible if the user has created an application and it contains REST APIs. This page is show in figure

[32] HAML language - http://haml.info/
[33] Bootstrap documentation - https://getbootstrap.com/docs/3.3/

5.2 and contains a documentation of all the endpoints available inside the application. It is formatted in a clean way using bootstrap's *<code>*, *<kbd>* and *<samp>*[34] tags such that it's readable for the users.

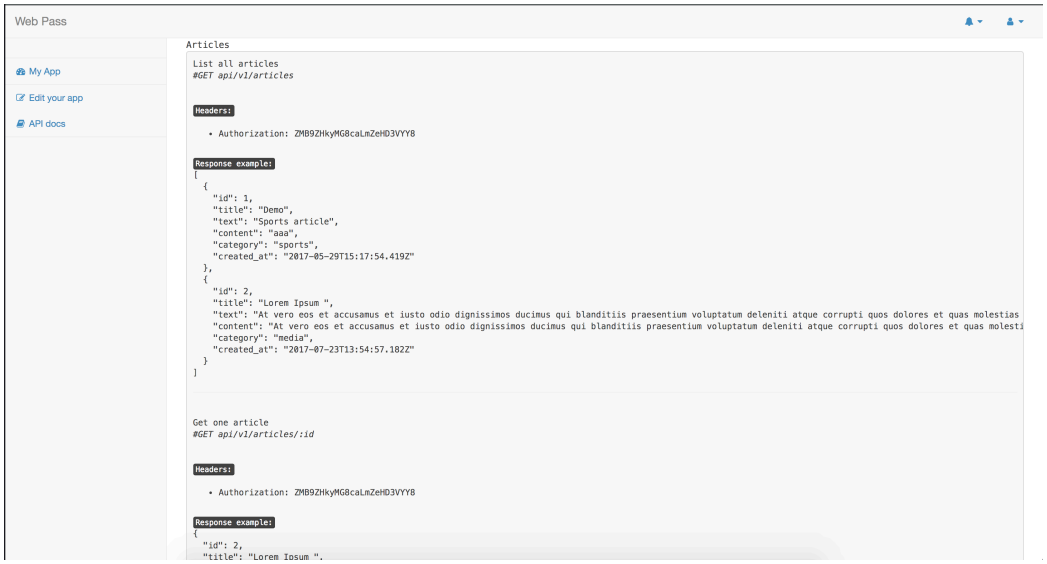


Figure 5.2 API Docs page

An important note about the graphical user interface is that in order to keep the code clean and not duplicate shared components, I've used Ruby on Rails's *ActionView::Layouts*[35] system which allows me to define layouts used on all the views and insert dynamic content specific to each page.

ActionView also allows me to define *partials* – which are files that can be rendered in any other file and can even render dynamic content since they can receive parameters. The partial's name must start with and underscore since they need to be distinguished from the regular views.

Such a partial was used to display the authorization headers on the API docs page, since they are requested at each request from the documentation as it can be seen in figure 5.2, this resulting in less and easier to maintain code. The GUIs main file structure is presented in figure 5.3 and it contains a view for each action available to the user after login and the previously mention partial file.

[34] Bootstrap code format - https://getbootstrap.com/docs/3.3/css/#code
[35] ActionView::Layouts - http://api.rubyonrails.org/classes/ActionView/Layouts.html

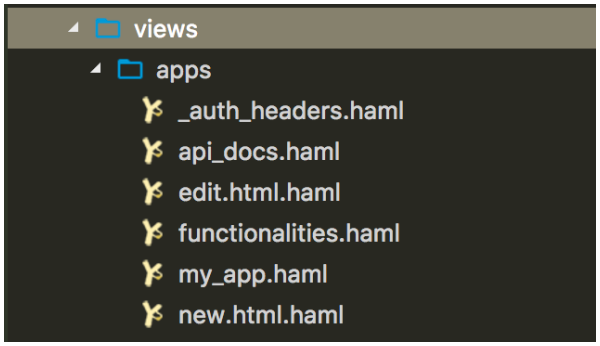


Figure 5.3 GUIs file structure

An important detail would be that the *My App* page is implemented using *WebSockets*, which means that each time the state of the application changes, the content of the page changes automatically. On each update of the application, a callback is run and it broadcasts the new object to the subscribers – web browser and it updates the content from *JavaScript.*

5.2. Architecture

In order to achieve logical separation of functionality we must consider layering of the application and setting boundaries and responsibilities between the modules. There are many benefits from layering the application, like scalability, flexibility the ability to improve performance of a particular component or the ability to replace some components without changing anything else from the application as a benefit of low coupling between the application.

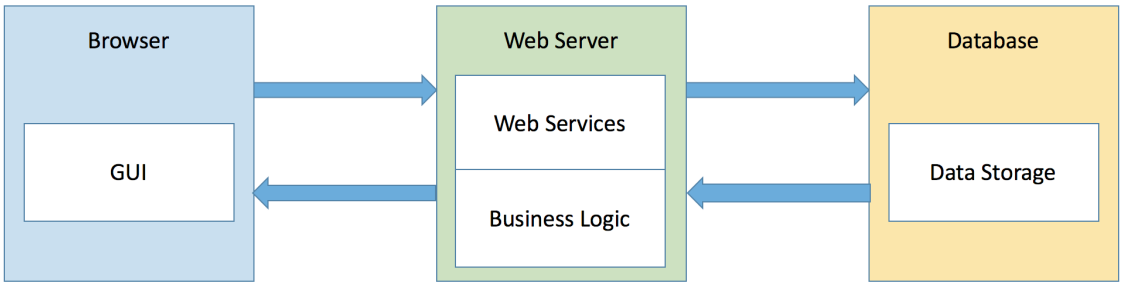


Figure 5.4 Application Architecture Overview

The figure 5.4 defines the main layers of the application, but a more detailed representation of the system is presented in figure 5.5.

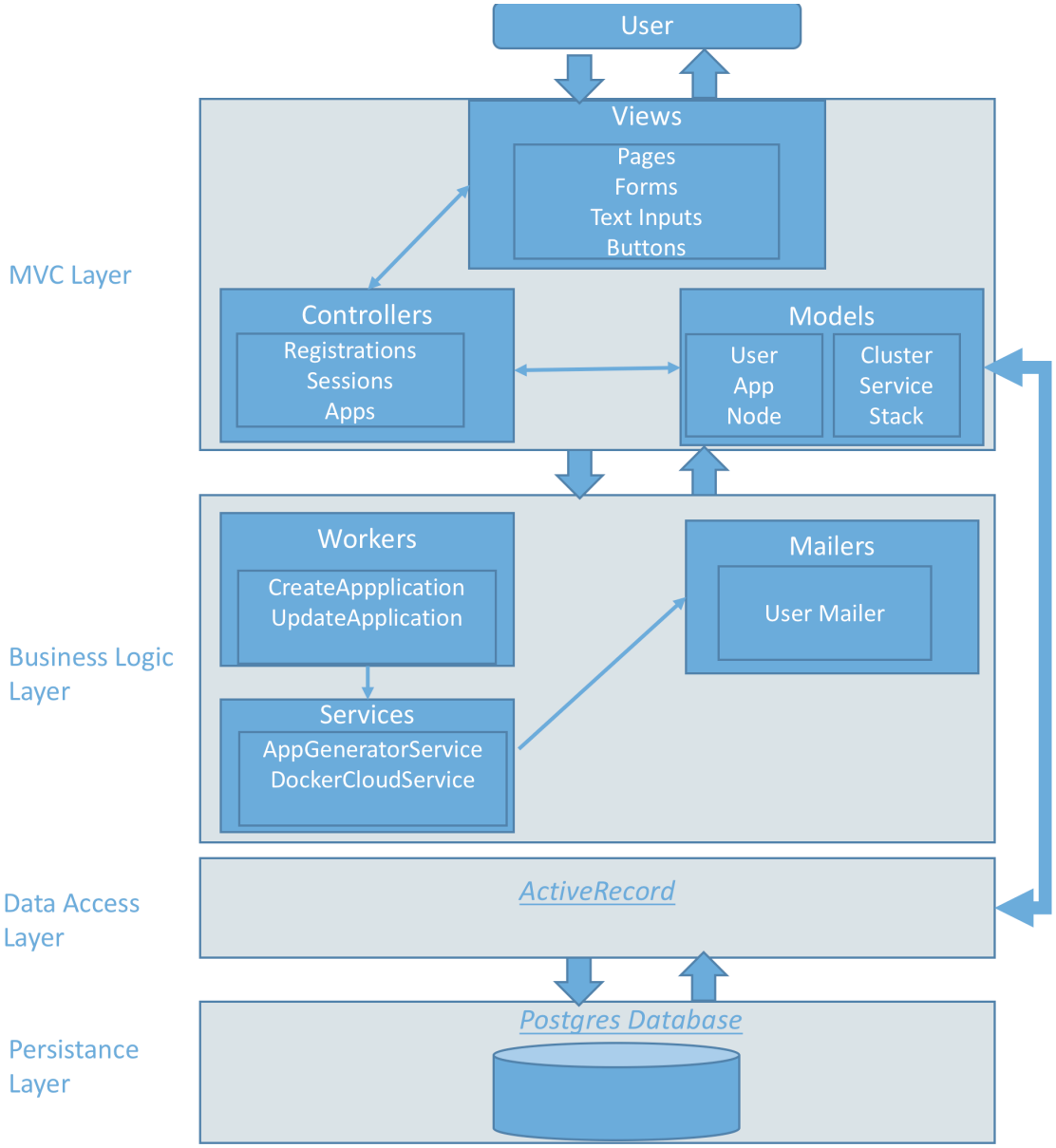


Figure 5.5 Platform Architecture Detailed

From figure 5.5 it can be noticed that the user interaction is handled by the *Model – View – Controller* components and the requests from the user are using *HTTP Protocols*[36].

The business logic in handled asynchronously and the communication is made using *redis*[37] to write the the jobs which are read from there by the workers and they start delegating jobs to different services. The solution with redis was chosen because the data is persisted, it's not using memory to store the parameters and the jobs to be run, but they are stored on the hard disk, which is a safer solution because it's a remanent memory,

[36] HTTP protocols - https://ro.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[37] Redis documentation - https://redis.io/documentation

unlike RAM memory. Also the hard – drive memory is a lot cheaper so it makes this communication protocol a suitable solution for the platform's business logic.

Besides the platform's architecture presented before, it is also important to present the UML Class diagram of the most important classes.

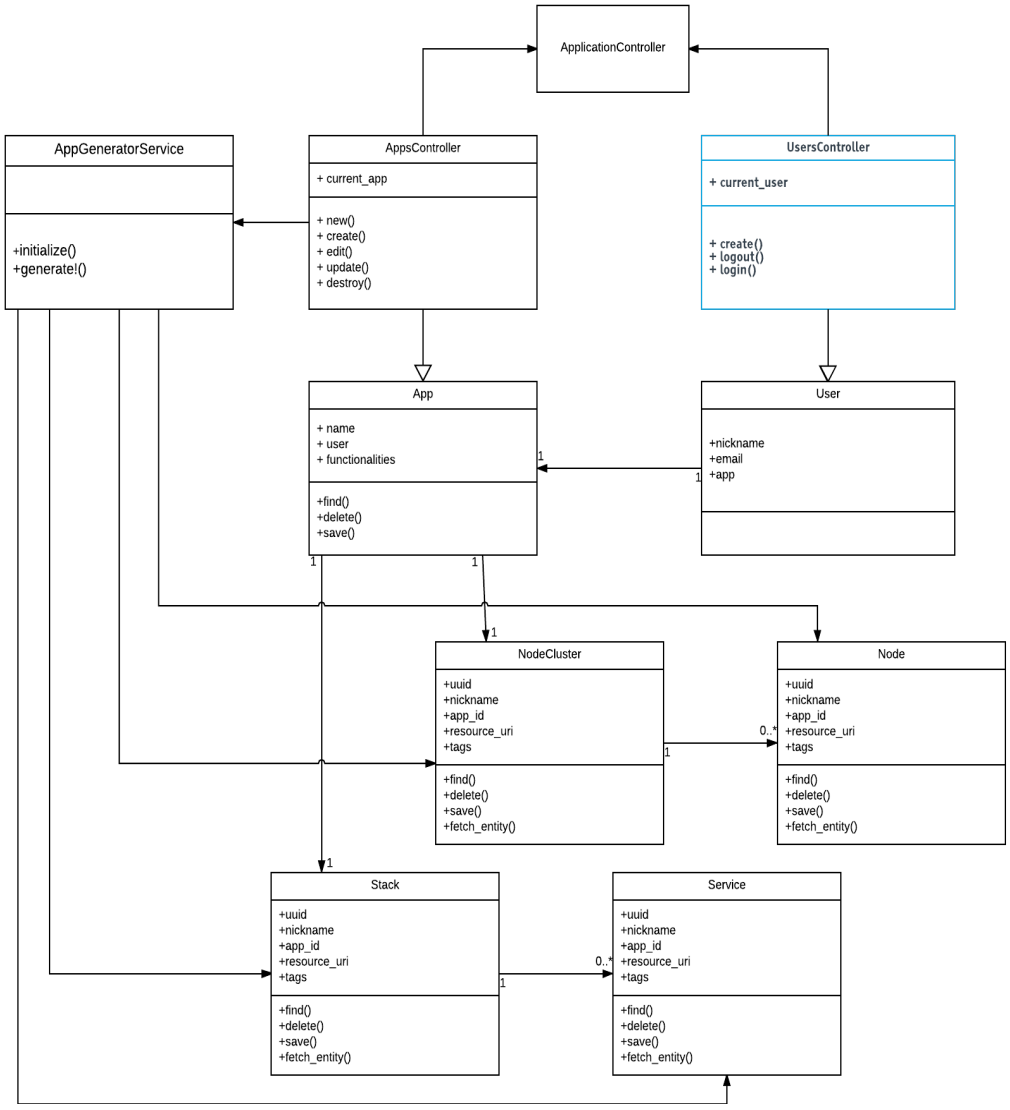


Figure 5.6 Platform's class diagram

It is presented in figure 5.6 describe the relations between the main classes, both from the *MVC* components and from the business logic. The *ApplicationController* is a base class and the *UsersController* and *AppsController* extends from there.

AppsController is responsible both for accessing the *App* model and for triggering an instance of the *AppGeneratorService* which is responsible for generating a new app and endsuring data persistency for the docker cloud entities, by using the models *NodeCluster, Node, Stack, App* and *Service*.

5.3. Database Design

The database used for this application is *PostgreSQL* version 9.4[38] and it's an alternative to *MySQL*[39]. The reason *PostgreSQL* was used instead of *MySQL* is because it's open source, supports more advanced features like *jsonb*[40] which was used in order to store the functionalities in the *app* table.

The tables, columns and relationships are described in figure 5.7.

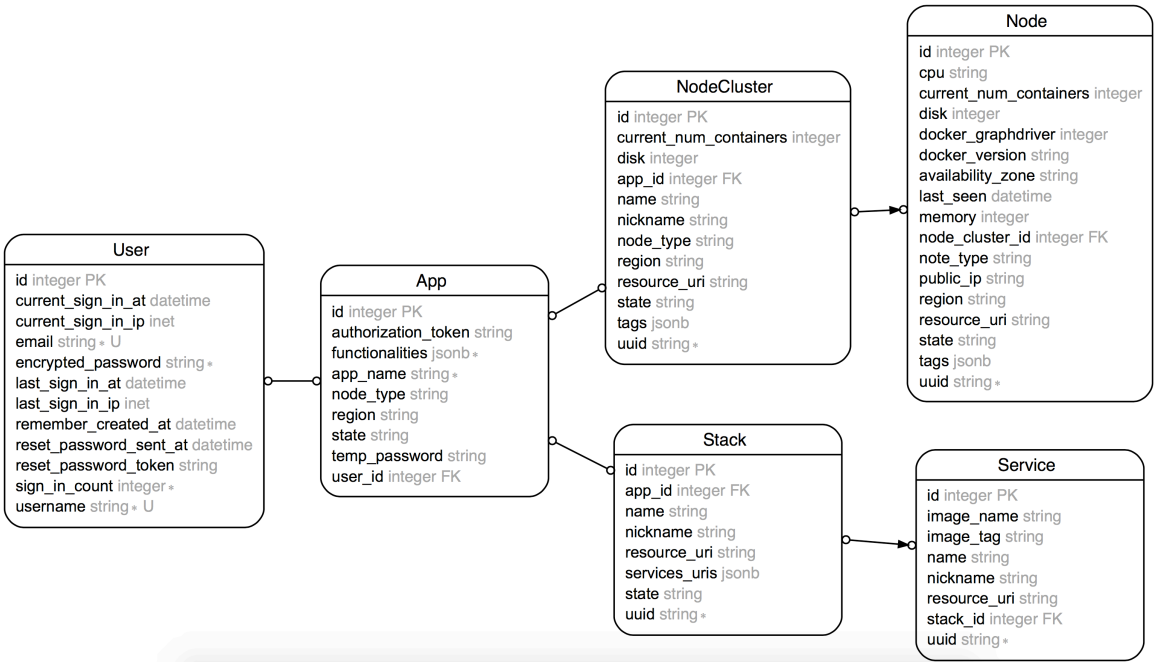


Figure 5.7 Database Design

The database is composed out of six tables: *User, App, NodeCluster, Node, Stack* and *Service*.

The *user* table is used to store information about the user and it has no foreign keys.

The *app* table store user's preferences for the app created. It is a one to one relationship between the *app* and *user* tables, the *app* table having the *user_id* as a foreign key.

The *NodeCluster, Node, Stack* and *Service* tables are created to persist the configuration and the data from the docker cloud service. The *NodeCluster* has many

[38] PostgreSQL 9.4 refrence- https://www.postgresql.org/docs/9.4/static/index.html
[39] MySQL - https://www.mysql.com/
[40] PostgresSQL jonb - https://www.postgresql.org/docs/9.4/static/functions-json.html

nodes hence the *node* table has a *node_cluster_id* foreign key. This is similar for the *stack* and *service* table, *service* having *stack_id* column as a foreign key. The *NodeCluster* and *Stack* tables have the *app_id* foreign key column.

The main constraints of the database are:

- Only one *app* entry for a *user_id* foreign key
- Only one *stack* entry for a *app_id* foreign key
- Only one *NodeCluster* entry for an *app_id* foreign key
- Uniqueness for *email* column in *user* table

5.4. Modules Implementation

5.4.1. Server Provisioning module – Docker Cloud Module

As stated in chapters three and four, docker offers many advantages for deploying and distributing a freshly generated web application. In consequence of that Docker and Docker Cloud services was used in order to generate and deploy the applications generated by the users.

In the platform, I've implemented a service called *DockerCloudService* which is a static class with stateless methods which are a middleware between the application and the docker API. This approach offers some advantages like low coupling, single responsibility and easy to reuse.

Next, we will describe the main methods of this service:

- *create_node_cluster(options = {})* – this method is used to create a new cluster of servers in which the application will run. It takes only one argument, which is optional and can set the name, the provider, the region, the memory, CPU, the number of servers and hard disk. Since the parameter is optional the module also has some defaults value – the region is in London, the provider is digital ocean and the servers have 1 GB of RAM and 60 GB of memory. The method returns either the cluster created or an error code if any errors occurred.
- *deploy_node_cluster(uuid)* – this method is used to deploy an existing cluster. The deployment process can take up to several minutes and for that reason, it's processed asynchronously so the method either returns an error message or the node cluster configuration. For those reasons the method can not be used to find out when the node cluster was deployed.
- *node_cluster(uuid)* – this method is used to query a requested node in order to get information about the status or configuration. It takes the uuid of the node_cluster as a parameter and returns an object with all the data available.
- *create_stack(options = {})* – this method is used to create a new stack(a collection of services) user by docker which contains all the microservices of the generated application. It takes one optional argument which

contains the stack's name and a list of services to run. The method returns either the stack created or an error message.

- *update_stack(uuid, options = {})* – this method is similar to the *create_stack* method, but it takes another parameter *uuid* in order to identify the stack. The second parameter is a hash which can contain the name or the services used by the stack.
- *deploy_stack(uuid)* – this method is used to deploy a previously created stack and all of it's services. The deployment process can take more time to be completed and for that reason it's processed asynchronous. In order to know when the process is completed the stack needs to be queried by a different method and read the status.

5.4.2. App Generator Service

The service *AppGeneratorService* is a class used when the users want to deploy a new application. It only has two public methods – the constructor(*initialize)* and a method called *generate!*. There are more private methods used and next we will take a closer look at the main methods.

- *initialize(app)* - the constructor method. Takes as an argument an instance of an *App* class. The constructor sets five instance variables: *app*, *user* – the user who created the application, *app_name* - a unique identifier , *functionalities* – an array of functionalities selected by the user, *temp_password* – a temporary password set for the user's newly generated application.
- *generate!* – a method which is orchestrating all the process of generating a new app by using the *DockerCloudService* and other private methods, also ensuring that they are called in the right order since some of the actions can't be parallelized.
- *create_node_cluster_record* – method used to create a *NodeCluster* record inside the database. This is done by using an *ActiveRecord* model called *NodeCluster* and it's necessary in order to ensure data persistency and caching the data from Docker.
- *deploy_node_cluster* – method is used to deploy a node cluster by using *DockerCloudService*'s method *depoloy_node_cluster* and also checking the status and allow executing the next operations when the deployment of all the nodes is complete.
- *create_and_deploy_stack* – a method used to create, add services and deploy a stack to docker. At first, the stack is created with nothing but the name of the app and an empty list of services, next a list of services is added to the stack and then the stack is created. This method also creates a *Stack* record in the database and it also stores all the services in the database in order to ensure data persistency.

- *notify_success* – method used to notify the user that the application has been successfully created and deployed. It is calling the *UserMailer*'s method *notify_success_deployment* which is sending an email to the user.

5.4.3. Update App Service

Service used for updating the application's configuration on Docker Cloud and also update the records in the database. It has only one static method called *perform* which has one argument *app_id.*

The first operation is to query the app from the database using an *App* model after that, the *app_service* is queried for that specific app since the service is the first think which can be updated – the functionalities. The next step is computing the new functionalities and updating the docker service and the record in the database, after that the app service is being redeployed. Next, the node's configuration is updated (RAM, location etc.) if the user changed any of the settings.

5.5. Application flow

This section describes how the objects and services interact in a time frame starting from a user action or trigger.

The *create application flow* is the main flow of the application as seen in figure 5.8 and it uses most of the services described in the previous chapters.

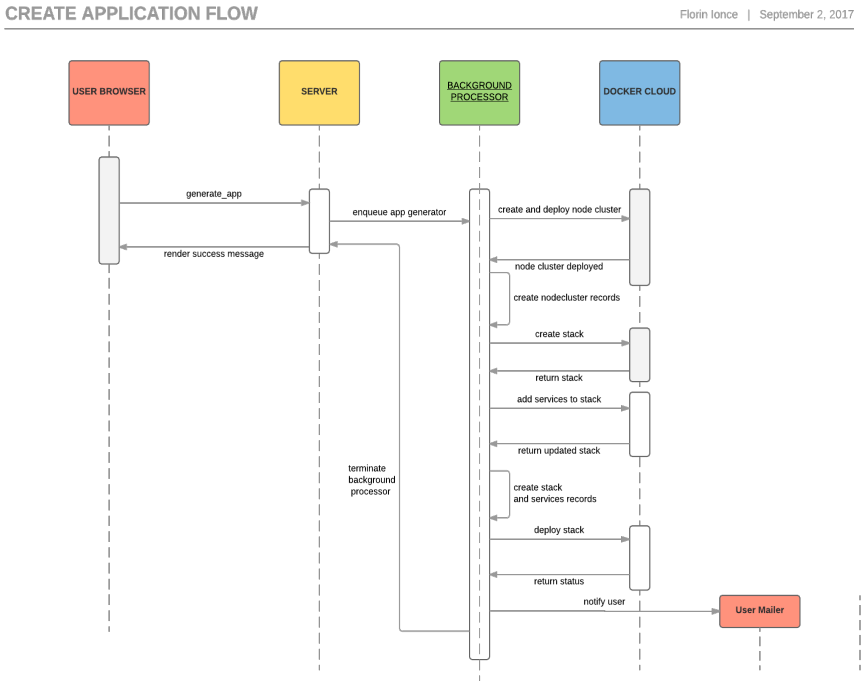


Figure 5.8 Create application sequence diagram

The event is triggered by the user and is represented in the first column and sent to the server. Since the rest of the flow is processed asynchronously the user receives a success message and the server enqueues a background process to generate the application. The background processor interacts with the docker cloud service and docker cloud API in order to create node clusters, stacks services and deploy them. They also interact with the models in order to store data in the database for persistency.

5.6. Generated application

5.6.1. Overview

In the previous sections of chapter 5 we've covered the implementation details of the platform and now we will describe the implementation of the generated application.

The generated application was also written using Ruby on Rails and it contains all the functionalities in forms of plugins. In order to use only the functionalities chosen by the user, the platform stores them in an instance variable and they are read in an initializer at boot time.

5.6.2. Implementation details

The generated application contains multiple microservices, so each user will have an application with microservices for the following:

- web application
- web server – nginx
- database
- database migrate

The *database*, *web application* and *web server* microservices have an auto-restart, mechanism in case of failure in order to ensure high availability. The *database migrate* microservice is used to migrate new changes of the database's structure and to ensure that all the scripts are run in the correct order – thus guaranteeing the proper setup and tables in the database.

In order to filter functionalities, the application contains *authorize_access(functionality)* which takes as a parameter called *functionality* and checks if that specific functionality was whitelisted by the user. If so, it will be available for the user to use and visible in the front-end, otherwise it will be hidden and if someone tries to access the resource from a web browser the route will not be found, the the user will be redirected to the root path and a flash message would appear saying the the

requested page does not exist. If a user tries to access an unauthorized resource, the API will return a message with status code 401 and a message:

```
{
   "error": "unauthorized",
   "status": 401
}
```

If the generated application contains an API, then all the requests require a header called *Authorization* with a value which is provided to the user on the platform both on the *My app* page and in the API's documentation.

If someone tries to access the API without the authentication header, the application responds with an unauthorized message and the status code 401.

An important feature of the boilerplate application is that it contains a function which runs when the app is first created and creates a user on the new database with the user's email from the platform and a temporary password generated by the *AppGeneratorModule.*

5.6.3. Examples

Generated application with *Articles* and *Events* functionality - a blog application where everyone is allowed to create and view events, and registered users are allowed to create articles, but unregistered users can only see them.

The figure 5.9 presents the events listing page, and it can be observed the the events have a location description and a time interval.

We can also see that the users have the option to create a new event which will be presented soon.

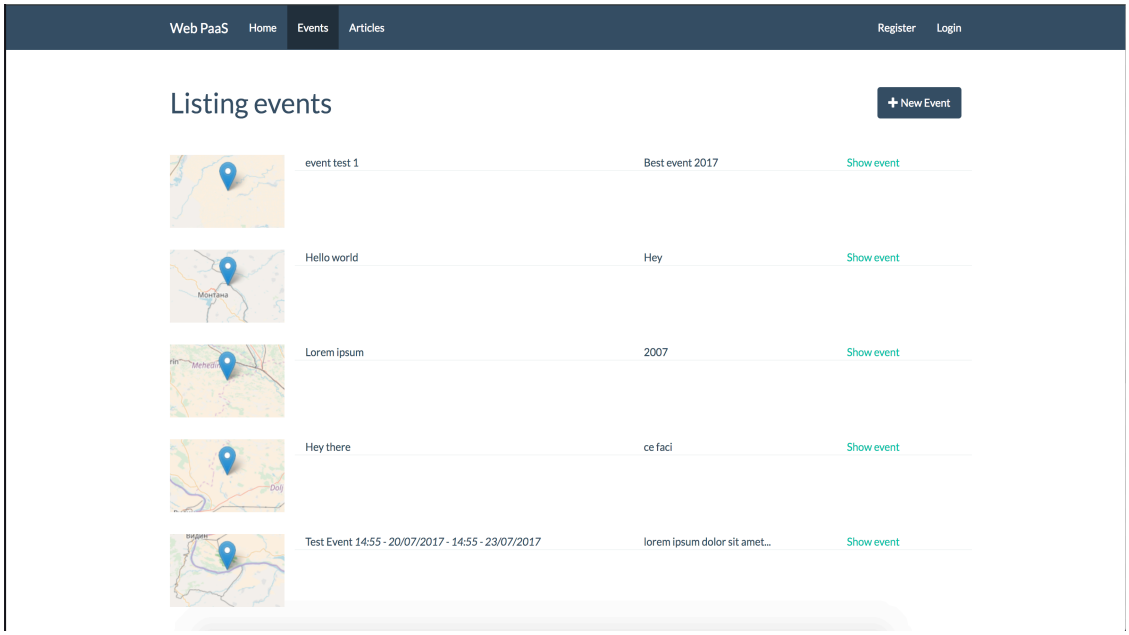


Figure 5.9 Event listing page

On the *Event show* (figure 5.10) we can notice that we have the same information provided for an event, with a larger map build with *leaflet* [41] with navigation capabilities and a full address which is automatically extracted from the latitude and longitude of the event with the help of the *geocoder*[42] library.

[41] Leaflet JS documentation - http://leafletjs.com/reference-1.2.0.html
[42] Geocoder - https://github.com/alexreisner/geocoder

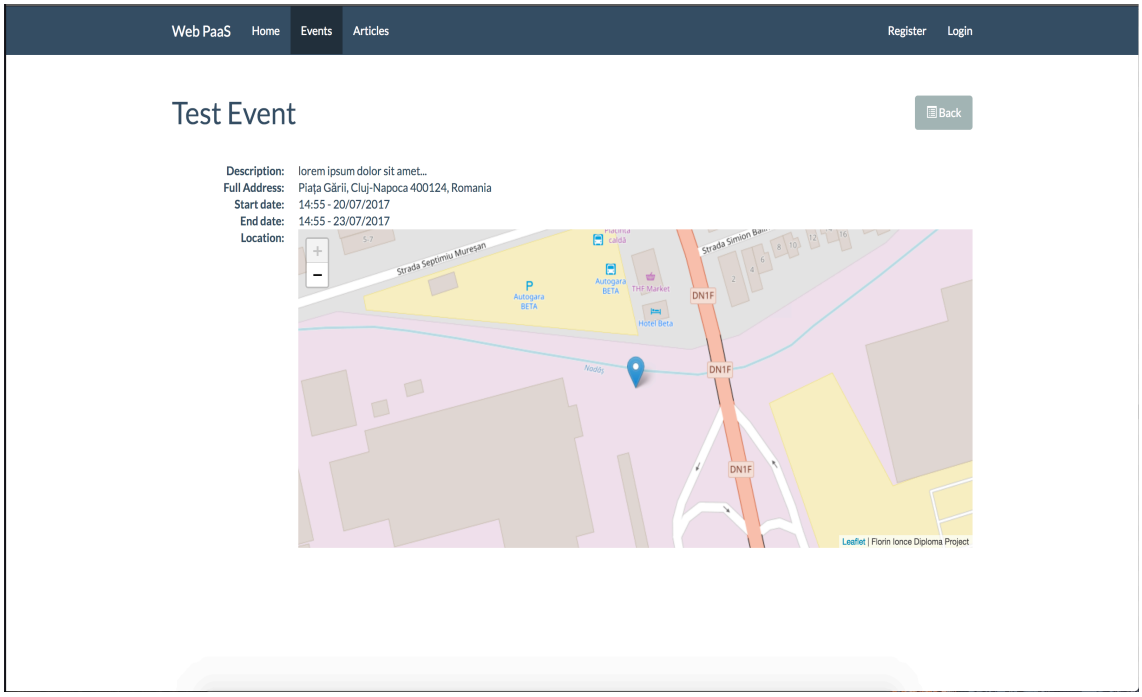


Figure 5.10 Event show page

Next we will se how a request is created using the generated REST API. The parameters can be sent either in JSON format, in which chase there is also required a header: *Content-Type: application/json* or as form data. This example will be using JSON data and it can be seen in figure 5.11 from Postman[43], both the request and the response.

Request parameters example:

```
{
        "event": {
            "title": "Test Event",
            "description": "lorem ipsum dolor sit amet...",
            "latitude": "46.7856304",
            "longitude": "23.5809621",
            "start_date": "2017-07-20T14:55:00.000Z",
            "end_date": "2017-07-23T14:55:00.000Z"
        }
}
```

[43] Postman tool - https://www.getpostman.com/

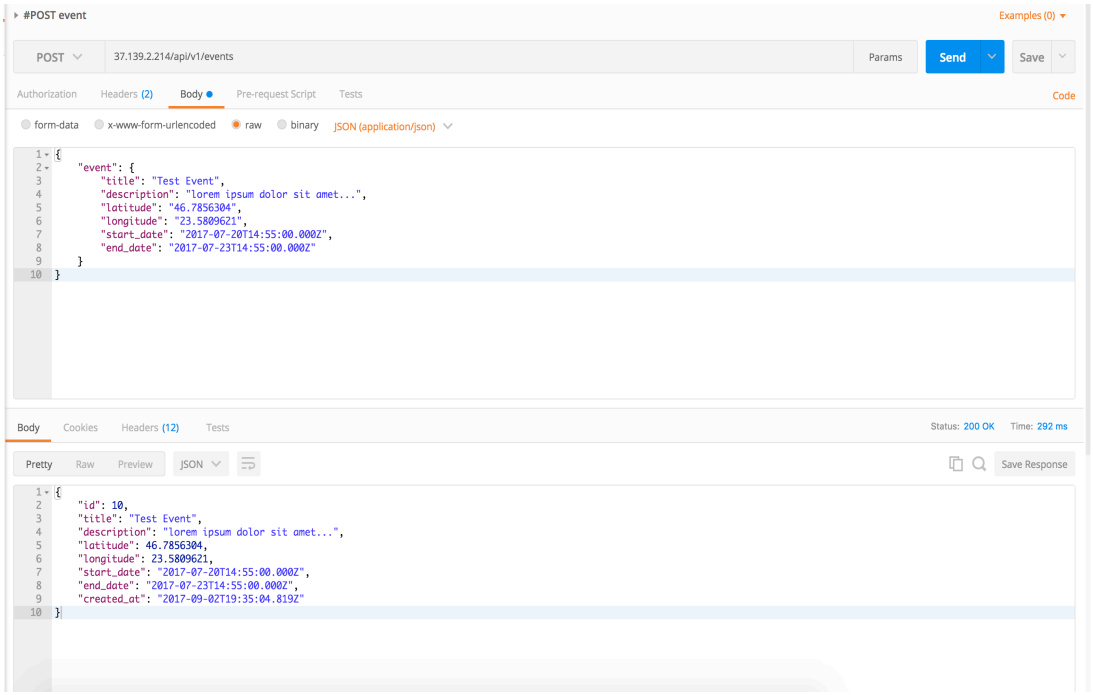


Figure 5.11 Postam example for creating an event

The response is an object in JSON format. An important note is that for all the API's responses on the event resources the response will be either similar or an array of similar objects.

Response:

```
{
   "id": 10,
   "title": "Test Event",
   "description": "lorem ipsum dolor sit amet...",
   "latitude": 46.7856304,
   "longitude": 23.5809621,
   "start_date": "2017-07-20T14:55:00.000Z",
   "end_date": "2017-07-23T14:55:00.000Z",
   "created_at": "2017-09-02T19:35:04.819Z"
}
```

It can be notice that the response contains the resource of the *ID, the title, the description, latitude, longitude, start date, end date* and the timestamp on when the resource was created. As a general rule, the API returns the same fields it can accept as parameters on the create action.

Chapter 6. Testing and Validation

Testing was made both by using unit testing with *rspec*[44] library and automated testing with *gatling*[45] which is designed to test the performance, response time, errors for a web server. It has out of the box support for HTTP protocols, and it can have scenarios defined which can be run with different settings like more user simultaneously on the website or more requests. In addition to those two methods, there was also the manual testing both of the platform and of the generated application.

6.1. Unit testing

As mentioned before, unit testing was made with rspec library and it was mainly focused on the platform's functionalities rather than the generated application's.

The unit tests cover all the main functionalities of the models related to the business logic and the *AppsController* as well as the *ApplicationController* from which the *AppsController* inherits from.

The reason we have 100% test coverage for this controllers is because they are handling the business logic of creating and updating an application – the main functionalities of this platform. The results were measured using *Simplecov*[46] and can be observed in figures 6.1 which shows an overall coverage of the controllers and figure 6.2 which shows detailed statistics about the *AppsController*.

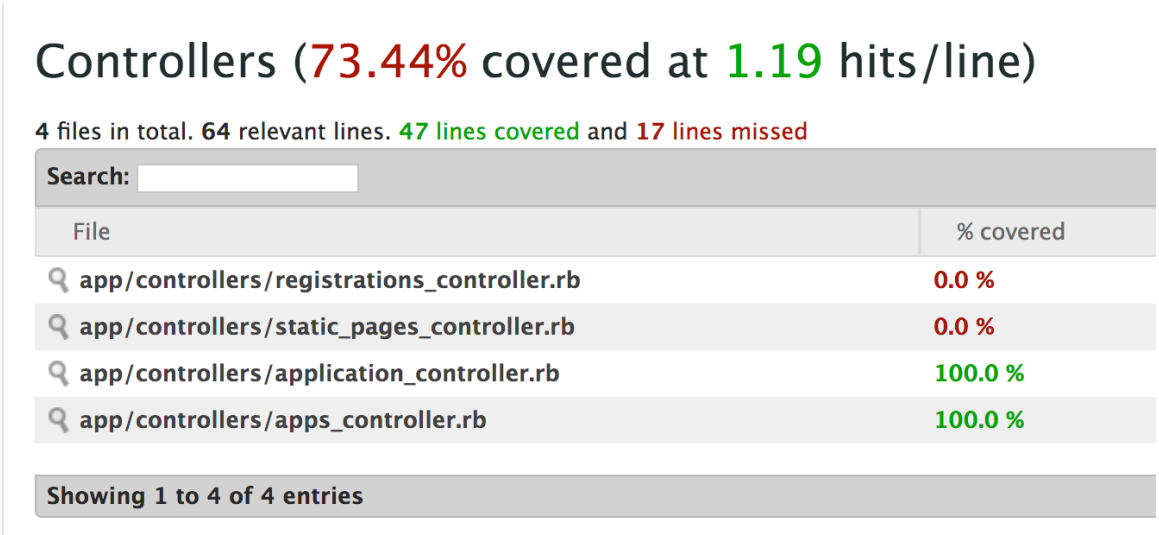


Figure 6.1 Controllers unit tests overview

[44] Rspec – testing library - https://github.com/rspec/rspec-rails
[45] Gatling testing tool - http://gatling.io/docs/current/
[46] Code Coverage tool - https://github.com/colszowka/simplecov

app/controllers/apps_controller.rb
100.0 % covered
41 relevant lines. 41 lines covered and 0 lines missed.

```ruby
# frozen_string_literal: true
class AppsController < ApplicationController
  before_action :authenticate_user!
  before_action :append_functionalities!, only: [:create, :update]
  before_action :set_app, only: [:edit, :update]

  def my_app
    @app = current_user.app
    render :_my_app_stats, layout: false if request.xhr?
  end

  def api_docs
    @app = current_user.app
  end

  def new
    @app = App.new
  end

  def create
    @app = App.new(app_params)
    @app.state = 'creating'
    if @app.save
      generate_app!

      redirect_to my_app_apps_path, notice: 'App was successfully created.'
    else
      render :new
    end
  end
```

app/controllers/apps_controller.rb

Figure 6.2 AppsController tests coverage

The controller tests cover all the scenarios in the process of creating applications and updating then, and when the users try to access the resources without having an account. Also, when creating or updating applications the tests also covers the scenarios when the user could submit wrong data or which would fain some validations or constraints from the database.

The models for *Apps, NodeClusters, Nodes, Stacks and Services* are also covered 100% as seen in figure 6.3 since they are ensuring the persistency and functionalities of the entire process.

app/models/node.rb	100.0 %
app/models/node_cluster.rb	100.0 %
app/models/service.rb	100.0 %
app/models/stack.rb	100.0 %

Figure 6.3 Main models test coverage

In addition to those, two of the main services were tested, the background processors used to create and generate application. Those results are presented in figure 6.4 and have a high testing coverage since a lot of the scenarios from the code are unlikely to happen.

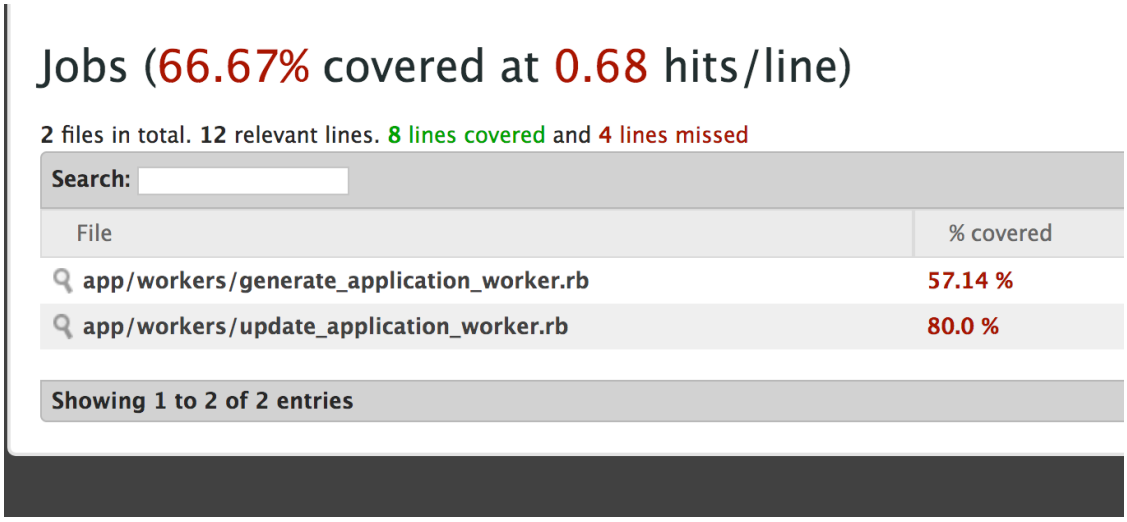


Figure 6.4 Background processors tests coverage

There are a total of 30 unit and integration tests written for the platform (figure 6.4) and they can be run easily by typing the command *bundle exec rspec spec/* - where *spec/* is the folder where the tests are placed.

In addition to that, each time a new commit is made on the platform, all the tests are run by Circle CI[47] - a continuous integration and delivery tool used on this platform so it ensures that each time a new build is created all the core functionalities still work as expected. An example of how the circle dashboard look like can be seen in figure 6.5 and 6.6 – showing an overview of the dashboard with a history of the latest builds.

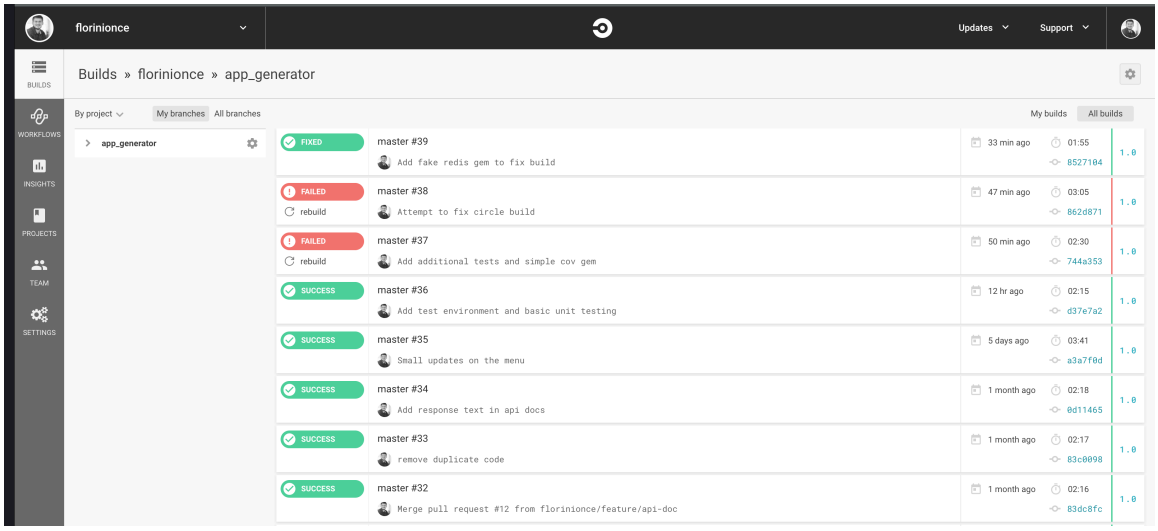


Figure 6.5 Circle CI dashboard

[47] Circle CI - https://circleci.com/

6.2. Automated testing

As mentioned before, the automated testing was made with Gotling tool it tests the generated application which is a modern tool with high capabilities and an excellent alternative to apache benchmark[48] with the possibility to record scenarios and run them in a certain order and with different number of simultaneously active users.

The way Gotling works is that it needs to record a scenario, by setting a proxy in the browser and inspecting all the requests that a user makes. After that it can redo the same scenario all over again with different setting in order to stress test the application.

The way we used Gotling here was that we created two applications with identical functionalities but with different node types – one app was using 1GB of ram memory (1CPU) and another one with 4GB (2 CPUs) of memory and see the difference between them.

After the tests were recorded, they were run and simulated 10000 users at the same time on the generated application, each of them performing multiple requests. The reason the simulation had this many users simultaneously was because it had to perform as a stress test in order to find the limits of the applications and see if the 4GB configuration was superior to the 1GB configuration.

The results show that the 4GB configuration is indeed superior, as we expected. The figures 6.6 and 6.7 show the overview charts for each configurations.

[48] Apache Benchmark - https://httpd.apache.org/docs/2.4/programs/ab.html

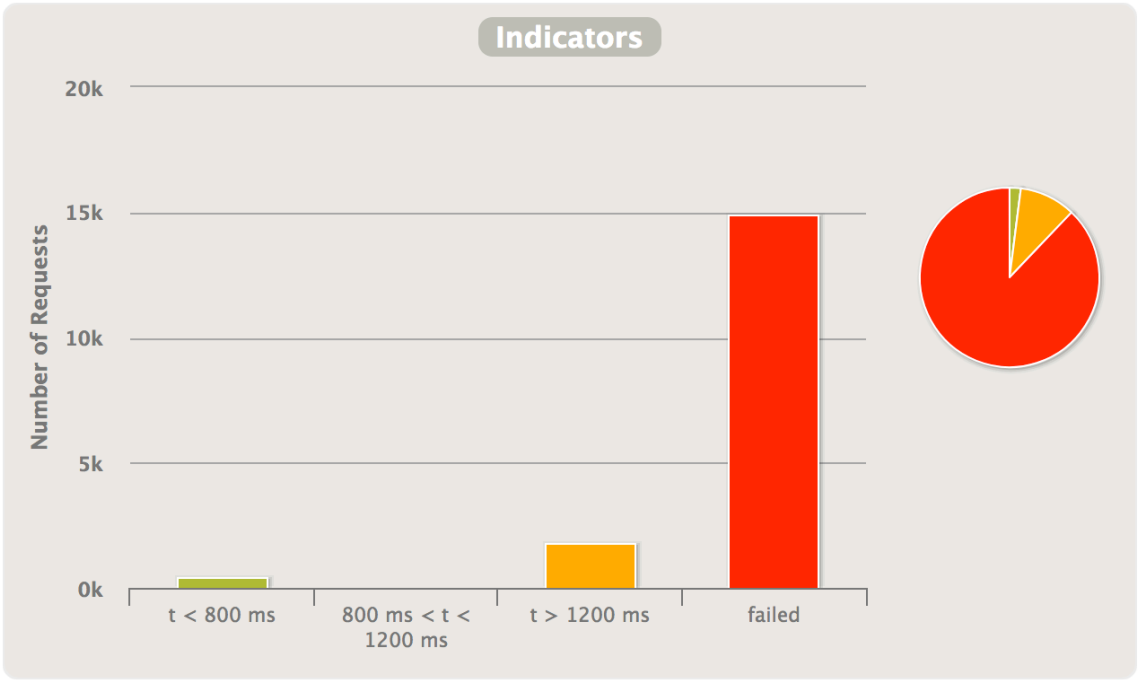


Figure 6.6 1GB app with 10000 users at once

It can be seen in figure 6.6 that the 1GB only managed to respond to 2217 requests from a total of 17112 which means it has a failure rate of 87%.

Figure 6.7 4GB app with 1000 users at once

As expected the 4GB configuration did a much better job, succeeding to respond to 6912 requests, having a failure rate of 57%, a lot better that the previous configuration.

This validates that the memory and server configurations are working and they are having a huge impact on the application's performance.

Chapter 7. User's manual

7.1. Installation

Run the following commands to install the application. The tutorial is made for Linux – Ubuntu operating system, but migrating to other operating systems should be similar considering that the application was developed on OSX.

7.1.1. Prerequirements

Install ruby and it's dependencies by running the following commands:

```
sudo apt-get install git-core curl zlib1g-dev build-essential libssl-dev libreadline-
dev libyaml-dev libsqlite3-dev sqlite3 libxml2-dev libxslt1-dev libcurl4-openssl-dev
python-software-properties libffi-dev tcl
sudo apt-get install python-psycopg2
sudo apt-get install libpq-dev
```

Install rbenv[49]

```
cd
git clone git://github.com/sstephenson/rbenv.git .rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bash_profile
echo 'eval "$(rbenv init -)"' >> ~/.bash_profile

git clone git://github.com/sstephenson/ruby-build.git ~/.rbenv/plugins/ruby-build
echo 'export PATH="$HOME/.rbenv/plugins/ruby-build/bin:$PATH"' >> ~/.bash_profile
source ~/.bash_profile
```

Install ruby and set it as global using rbenv

```
rbenv install -v 2.3.0
rbenv global 2.3.0

echo "gem: --no-document" > ~/.gemrc
```

Install bundler and rails

```
gem install bundler
gem install rails
rbenv rehash
```

Install javascript runtime dependencies (node js)

```
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
```

[49] rbenv - https://github.com/rbenv/rbenv

```
sudo apt-get install nodejs
```

Install postgresql and create a database role

```
sudo apt-get install postgresql postgresql-contrib

sudo -i -u postgres

createuser --interactive
exit
```

Set a password for the user (recommended)

```
ALTER USER user PASSWORD 'newPassword';
\q
```

Install redis

```
cd /tmp
curl -O http://download.redis.io/redis-stable.tar.gz
tar xzvf redis-stable.tar.gz
cd redis-stable
make
make test
sudo make install
sudo mkdir /etc/redis
sudo cp /tmp/redis-stable/redis.conf /etc/redis
sudo nano /etc/redis/redis.conf

sudo adduser --system --group --no-create-home redis
sudo mkdir /var/lib/redis
sudo chown redis:redis /var/lib/redis
sudo chmod 770 /var/lib/redis
sudo systemctl start redis
sudo systemctl enable redis
```

Set environment variables required by the application – for this a valid DockerCloud account is needed in order to set the username and the password

```
rake secret
```

And copy the output and paste it at the end of the next command

```
export SECRET_KEY_BASE=
```

Store Docker Cloud credentials to environment variables

```
export DOCKER_CLOUD_USERNAME=username
export DOCKER_CLOUD_PASSWORD=password
```

7.1.2. Install the application

First step is to clone the application

```
git clone git@github.com:florinionce/app_generator.git
```

Installing the application

```
bundle install
bundle exec rake db:create
bundle exec rake db:migrate
```

Running the application – in order for the application to run, it needs to have two process started in parallel – the rails application and the sidekiq[50] processor – in order to process some processes in the background.

```
bundle exec rails server
bundle exec rails sidekiq
```

7.2. User manual

In this section we will present what steps should the user take in order to use the application. The application is straightforward to use since one of the non functional requirements was to be user friendly.

Accessing the first page will show two buttons on the left side and the user should click Sign Up in order to create an account, or Sign in in order to login if it's an already registered user.

The sign up form is presented in figure 7.1 and has basic registration fields. Once the user submits the form, it is automatically redirected to the root path of registered users (figure 7.2), from where the user can click create new application.

[50] Sidekiq background processor - https://github.com/mperham/sidekiq

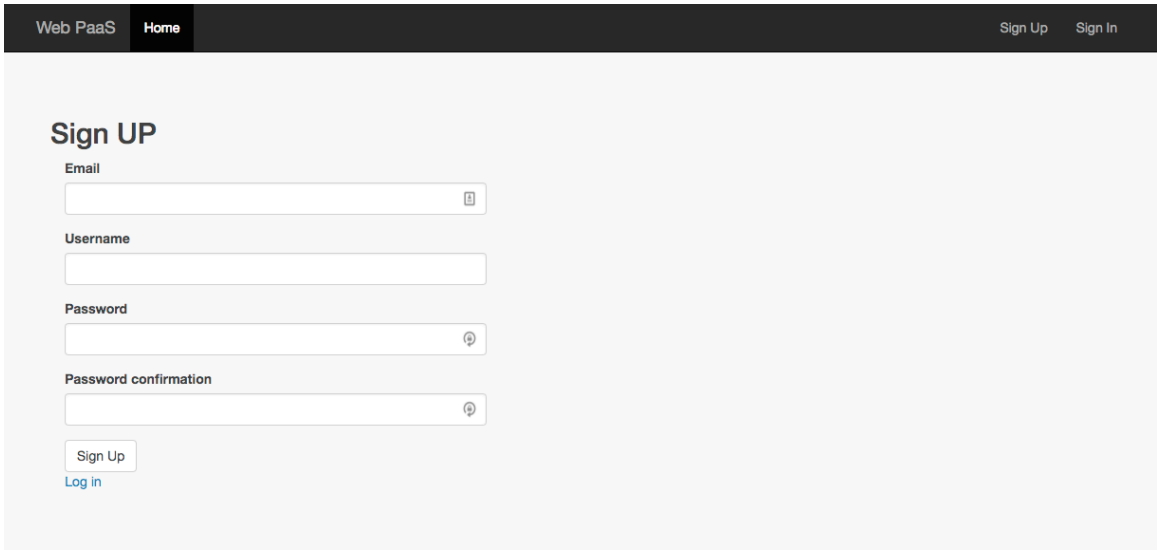


Figure 7.1 Sign up page.

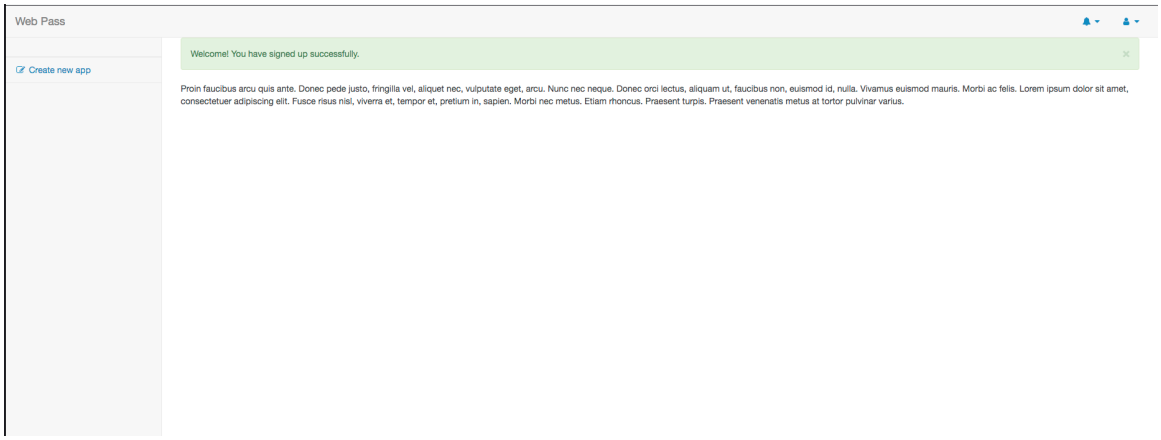


Figure 7.2 Registered root path

The *create new app* page presented in figure 7.3 contains some basic field, including the application name, dropdowns for each functionality from which it can select if it's a desired functionality, and how to use it – Web, API or both.

The last inputs are used to set the region of the application and the server capacity in order to fulfill the requirements.

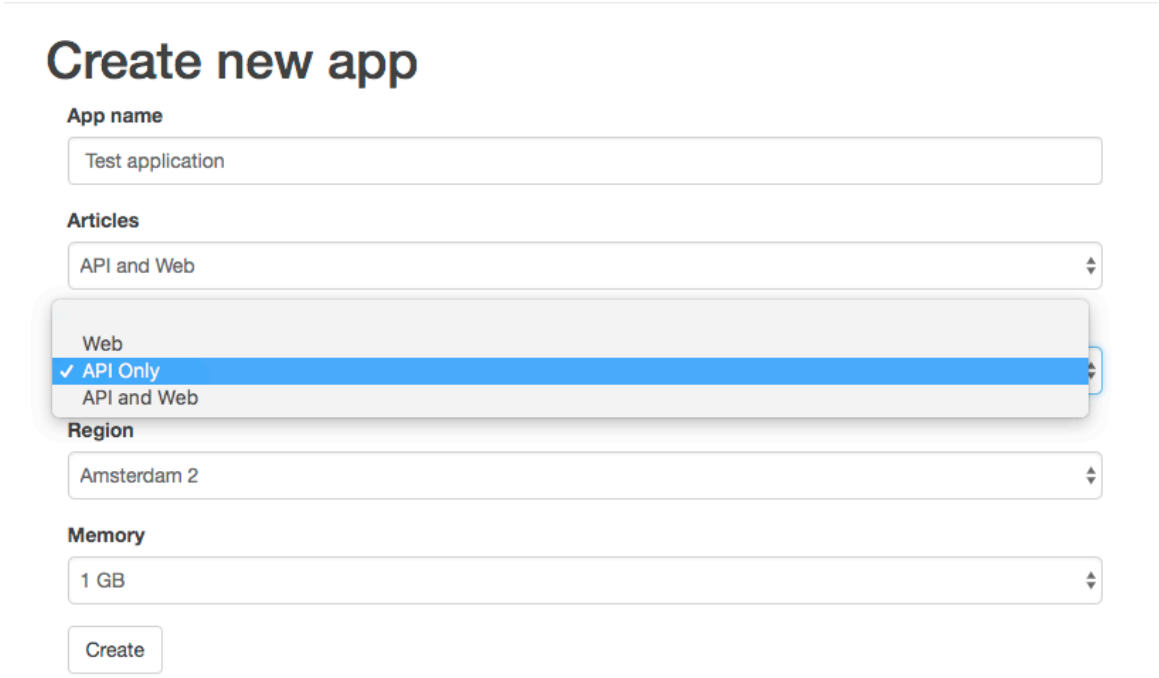


Figure 7.3 Create application page

Once the application is created, the user is redirected to *my app* page where some information about the application is available, together with links to the edit page, API documentations page. Also, once the app is deployed a link to the application will be available.

This page shown in figure 7.4 refreshes automatically so the user does not have to continuously check the deployment's process status.

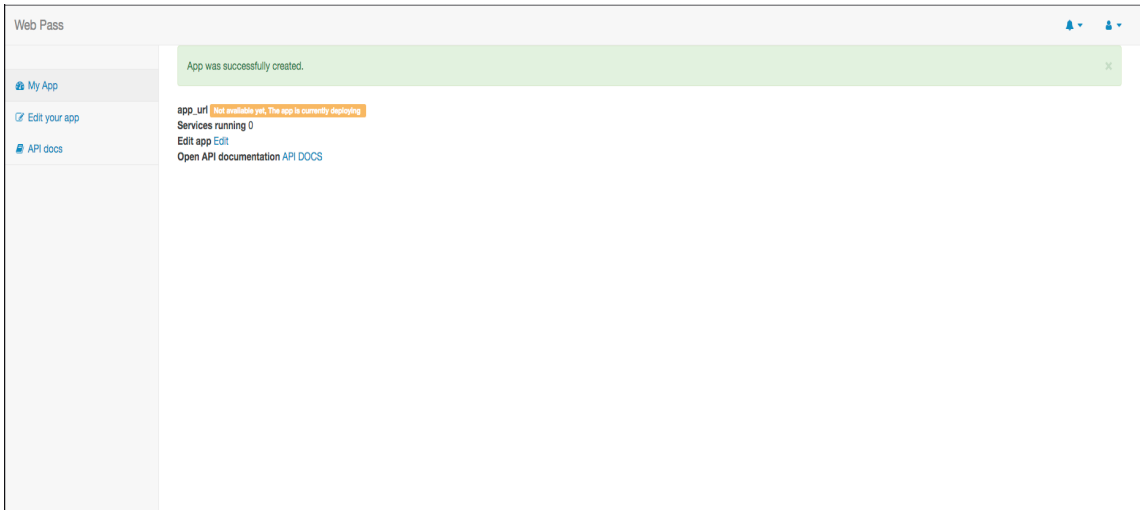
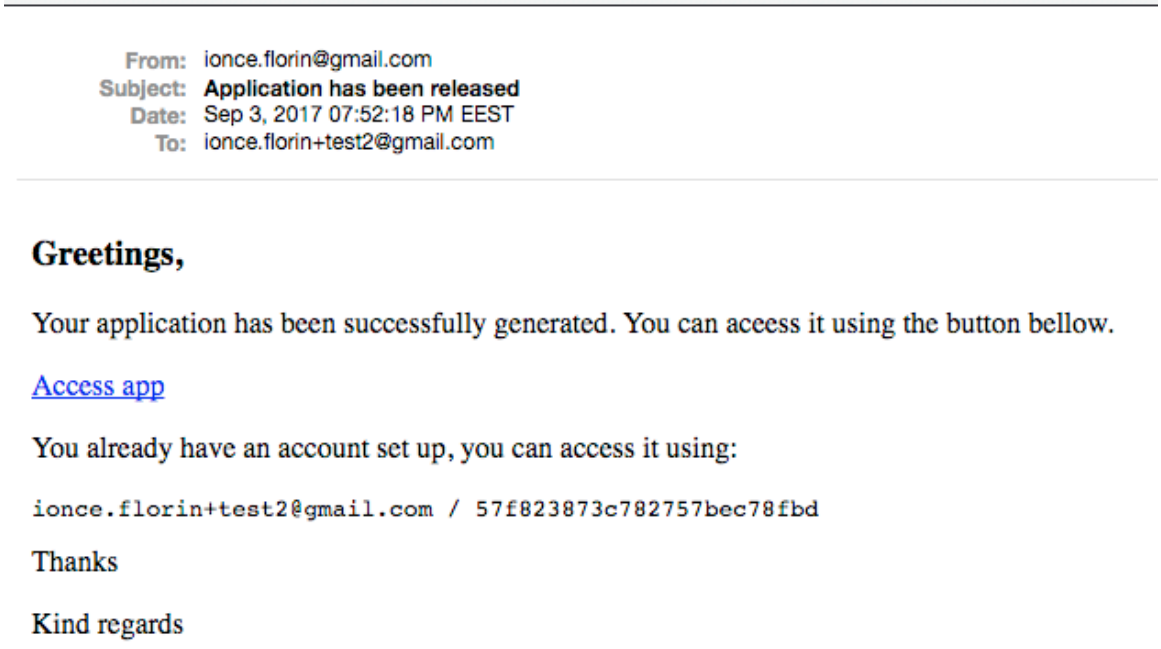


Figure 7.4 My app page

Once the application is deployed, the user also receives an email shown in figure 7.4 with a url to the generated application and the access credentials.



From: ionce.florin@gmail.com
Subject: **Application has been released**
Date: Sep 3, 2017 07:52:18 PM EEST
To: ionce.florin+test2@gmail.com

Greetings,

Your application has been successfully generated. You can aceess it using the button bellow.

Access app

You already have an account set up, you can access it using:

`ionce.florin+test2@gmail.com / 57f823873c782757bec78fbd`

Thanks

Kind regards

Figure 7.5 Notify email received by the user

In addition to the presented scenarios, the users can edit the application, the page is similar to the *create new application* page, presented in figure 7.3. Also the users can access the API documentation page accessible from the side menu or from the *my app* page and it contains information about all he endpoints available.

Chapter 8. Conclusions

Nowadays, people are using more and more web and mobile applications and the knowledge required to create a web application or building the platform and the back-end for a mobile or IoT application can be a problem for non-technical users. From this reasons, a platform capable of generating dynamic web applications and APIs ca become very helpful.

8.1. Current Results

The main goals and core functionalities of the platform were achieved and implemented:

- The ability to create web applications dynamically without coding
- The ability to create REST APIs dynamically without coding
- The application is hosted on the cloud on a dedicated server
- The ability to scale the application
- Automatic deployment in cloud
- The ability to update the application's functionalities
- Data persistency
- Some basic functionalities available on the platform
- The ability to easily add more functionalities available for the users
- The ability to register
- Email notifications for the users
- API documentation for the generated endpoints

8.2. Further Development

The application is in an initial phase, and its main purpose was to prove that it's possible to generate and deploy applications dynamically in cloud and from that reason it lacks other features which can be added in the future.

Main improvements available:

- The generated application cloud be able to scale automatically based on the user load.
- Adding social login to the platform
- Adding more functionalities available to the users in forms of plugins
- Generate analytics available to the users based on the traffic they get on the application
- Offer an API which allows anyone to create plugins for the platform
- Create database backups on each of the application
- Offer uptime monitoring and alerts

- Add alerts when the application reaches a certain load percentage
- Improve user experience

Bibliography

[1] K. J. Saila, "UbiQloud: A Platform as a Service for the Web of Things", Aalto University, Masters Degree, Helsinki, 2012.

[2] G. Nalin, "Orchestration of smart objects with MQTT for the Internet of Things", University of Padua, Department of Information Engineering Master Degree in Computer Engineering, 2014

[3] Bc. Marek Jelen, "Platform for deploying web applications", Masarykova Univerzita Fakulta Informatiky, Brno, 2011

[4] O.S. Tezer, How To Scale Ruby on Rails Applications Across Multiple Droplets, https://www.digitalocean.com/community/tutorials/how-to-scale-ruby-on-rails-applications-across-multiple-droplets-part-1, 2014

[5] Number of internet users worldwide from 2005 to 2015, http://www.statista.com/statistics/273018/number-of-internet-users-worldwide/

[6] Types of cloud computing, https://aws.amazon.com/types-of-cloud-computing/

[7] Ilya Sabanin, "Deployments Best Practices", http://guides.beanstalkapp.com/deployments/best-practices.html.

[8] How Heroku works, https://devcenter.heroku.com/articles/how-heroku-works.

[9] Fink, John. Docker: a software as a service, operating system-level virtualization framework. Code4Lib Journal, 2014.

[10] Turnbull, James. "The Docker Book: Containerization is the new virtualization", 2014.

[11] Docker cloud API reference, https://docs.docker.com/apidocs/docker-cloud/

[12] Srirama, Satish. Scaling Applications on the Cloud, https://courses.cs.ut.ee/MTAT.08.027/2016_spring/uploads/Main/L3_ScaleOnCloud2016.pdf , 2016.

[13] Chris, Stump, Docker for an Existing Rails application, 2016, http://chrisstump.online/2016/02/20/docker-existing-rails-application/

[14] Middleton, Neil; Schneeman, Richard. Heroku: Up and Running: Effortless Application Deployment and Scaling. " O'Reilly Media, Inc.", 2013.

[15] Docker engine, https://docs.docker.com/engine/api/

[16] Digital ocean API, https://developers.digitalocean.com/documentation/v2/

[17] Ortiz, Ariel. “An introduction to metaprogramming. Linux Journal”, 2007.

[18] Perrotta, Paolo. “Metaprogramming Ruby”. Pragmatic Bookshelf, 2010.

[19] Web server performance comparison, https://help.dreamhost.com/hc/en-us/articles/215945987-Web-server-performance-comparison

Appendix 1. Glossary

Term	Definition
API	An application programming interface (API) is a particular set of rules ('code') and specifications that software programs can follow to communicate with each other. It serves as an interface between different software programs and facilitates their interaction, similar to the way the user interface facilitates interaction between humans and computers.
RESTful	Representational State Transfer is an architectural style, and an approach to communications that is often used in the development of Web services.
Web application deployment	The process of copying, installing the web application on a server or in cloud and exposing it such that it can be accessed by the targeted users.
HTTP Protocols	Most used protocol to access information via internet – World Wide Web.
Microservices	Service-oriented architectural style which structures an application into a collection of loosely coupled services.
Monolithic architecture	Architectural design for software applications designed to be self-contained with interconnected components.

Appendix 2. Figures and Tables list

Table 1 Similar systems conclusions 15

Figure 2.1 Basic use case diagram 4
Figure 3.1 Cloud computing models 7
Figure 3.2 Infrastructure as a service layer[2] 7
Figure 3.3 Platform as a service layer[2] 8
Figure 3.4 Software as a service layer[2] 8
Figure 3.5 Comparison between virtual machines and docker containers[6] 10
Figure 3.6 Docker vs KVM boot time measured in seconds[6] 10
Figure 3.7 CPU computing power[6] 11
Figure 3.8 Prismic content configuration and output preview 13
Figure 3.9 Building a squarespace page 14
Figure 4.1 Basic platflorm flow 16
Figure 4.2 High level project architecture 16
Figure 4.3 Server provider module state diagram 18
Figure 4.4 Register / Login mock-up 19
Figure 4.5 Application generator module state diagram 21
Figure 4.6 Conceptual architecture 26
Figure 4.7 Functionalities page mock-up 27
Figure 4.8 Create your application page 28
Figure 5.1 Create new application page 33
Figure 5.2 API Docs page 34
Figure 5.3 GUIs file structure 35
Figure 5.4 Application Architecture Overview 35
Figure 5.5 Platform Architecture Detailed 36
Figure 5.6 Platform's class diagram 37
Figure 5.7 Database Design 38
Figure 5.8 Overall framework architecture 29
Figure 5.9 Create application sequence diagram 41
Figure 5.10 Event listing page 44
Figure 5.11 Event show page 45
Figure 5.12 Postam example for creating an event 46
Figure 6.1 Controllers unit tests overview 47
Figure 6.2 AppsController tests coverage 48
Figure 6.3 Main models test coverage 48
Figure 6.4 Background processors tests coverage 49
Figure 6.5 Circle CI dashboard 49
Figure 6.6 1GB app with 10000 users at once 51
Figure 6.7 4GB app with 1000 users at once 51
Figure 7.1 Sign up page. 56
Figure 7.2 Registered root path 56
Figure 7.3 Create application page 57
Figure 7.4 My app page 57

Figure 7.5 Notify email received by the user..... 58